

**Oracle8i**

Application Developer's Guide - Fundamentals

Release 2 (8.1.6)

December 1999

Part No. A76939-01

**ORACLE**

---

Application Developer's Guide - Fundamentals, Release 2 (8.1.6)

Part No. A76939-01

Copyright © 1996, 1999, Oracle Corporation. All rights reserved.

Primary Author: John Russell

Contributing Authors: M. Bauer, M. Cyran, J. Gibb, G. Gonzalez, V. Krishnamurthy, M. Krishnaprasad, J. Melnick, R. Moran, D. Raphaely, R. Smith, R. Urbano

Contributors: D. Alpern, A. Amor, G. Arora, V. Arora, J. Basu, R. Baylis, E. Beldin, S. Chandrasekar, T. Chang, A. Chaudry, W. Creekbaum, D. Das, M. Davidson, G. Doherty, J. Draaijer, B. Goyal, M. Hartstein, J. Haydu, K. Jacobs, M. Jaganath, N. Jain, H. Jakobsson, A. Jasuja, R. Jenkins Jr., R. Kasamsetty, J. Klein, R. Kooi, S. Krishnamurthy, R. Krishnan, S. Krishnaswamy, P. Lane, N. Le, C. Lei, L. Leverenz, J. Loaiza, D. Lorentz, W. Maimone, D. McMahon, A. Mendelsohn, M. Moore, R. Murthy, K. Muthiah, K. Muthukkaruppan, R. Narayanan, T. Nhu Bui, V. Nimani, T. Portfolio, M. Pratt, S. Puranik, T. Pystynen, M. Ramacher, S. Samu, U. Sangam, A. Sethi, P. Shah, N. Shariatpanahy, T. Smith, J. Srinivasan, S. Subramanian, U. Sundaram, D. Surber, S. Suri, N. Tang, J. Tsai, A. Tsukerman, S. Urman, P. Vasterd, G. Viswana, W. Wang, D. Wong, B. Wright, R. Yaseen

Graphic Designer: V. Moore

The Programs (which include both the software and documentation) contain proprietary information of Oracle Corporation; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Oracle Corporation.

If the Programs are delivered to the U.S. Government or anyone licensing or using the programs on behalf of the U.S. Government, the following notice is applicable:

**Restricted Rights Notice** Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and Oracle Corporation disclaims liability for any damages caused by such use of the Programs.

Oracle is a registered trademark, and Pro\*Ada, Pro\*COBOL, Pro\*FORTRAN, SQL\*Loader, SQL\*Net, SQL\*Plus, Designer/2000, Developer/2000, Net8, Oracle Call Interface, Oracle7, Oracle8, Oracle8i, Oracle Forms, Oracle Parallel Server, PL/SQL, Pro\*C, Pro\*C/C++ and Trusted Oracle are trademarks or registered trademarks of Oracle Corporation. All other company or product names mentioned are used for identification purposes only and may be trademarks of their respective owners.

---

---

# Contents

<b>Send Us Your Comments .....</b>	<b>xvii</b>
<b>Preface.....</b>	<b>xix</b>
<b>Part I Introduction To Working With The Server</b>	
<b>1 Understanding the Oracle Programmatic Environments</b>	
<b>Overview of PL/SQL .....</b>	<b>1-2</b>
How Does PL/SQL Work?.....	1-2
Advantages of PL/SQL .....	1-3
<b>Overview of OCI.....</b>	<b>1-7</b>
Advantages of OCI.....	1-7
Parts of the OCI.....	1-8
Procedural and Non-Procedural Elements.....	1-8
Building an OCI Application .....	1-9
<b>Overview of Oracle Objects for OLE .....</b>	<b>1-11</b>
The OO4O Automation Server .....	1-12
OO4O Object Model.....	1-12
Support for Oracle LOB and Object Datatypes.....	1-17
The Oracle Data Control.....	1-19
The Oracle Objects for OLE C++ Class Library .....	1-19
Additional Sources of Information .....	1-19
<b>Overview of Pro*C/C++.....</b>	<b>1-20</b>
How You Implement a Pro*C/C++ Application.....	1-20

Highlights of Pro*C/C++ Features .....	1-21
New Oracle8i Features Supported .....	1-22
<b>Overview of Pro*COBOL .....</b>	<b>1-23</b>
How You Implement a Pro*COBOL Application .....	1-23
Highlights of Pro*COBOL Features .....	1-24
New Oracle8i Features Supported .....	1-25
<b>Overview of Oracle JDBC .....</b>	<b>1-26</b>
JDBC Thin Driver .....	1-26
JDBC OCI Driver .....	1-26
The JDBC Server Driver .....	1-27
Extensions of JDBC .....	1-27
Sample Program for the JDBC Thin Driver .....	1-27
Java in the RDBMS .....	1-29
Why Use Stored Procedures? .....	1-29
JDBC in SQLJ Applications .....	1-30
<b>Overview of Oracle SQLJ .....</b>	<b>1-30</b>
SQLJ Tool .....	1-30
SQLJ Design Goals .....	1-31
Strengths of Oracle's SQLJ Implementation .....	1-32
Comparison of SQLJ with JDBC .....	1-32
SQLJ Example for Object Types .....	1-33
SQLJ Stored Procedures in the Server .....	1-36
<b>Choosing a Programming Environment .....</b>	<b>1-37</b>
Use OCI or a Precompiler? .....	1-37
Built-in Packages and Libraries .....	1-38
Java vs. PL/SQL .....	1-39

## Part II Designing the Database

### 2 Managing Schema Objects

<b>Managing Tables .....</b>	<b>2-2</b>
Designing Tables .....	2-3
Creating Tables .....	2-4
Altering Tables .....	2-9
Dropping Tables .....	2-10

<b>Managing Temporary Tables</b> .....	2-11
Creating Temporary Tables .....	2-12
Using Temporary Tables .....	2-12
Examples: Using Temporary Tables .....	2-13
<b>Managing Views</b> .....	2-15
Creating Views .....	2-15
Replacing Views .....	2-17
Using Views .....	2-18
Dropping Views .....	2-20
<b>Modifying a Join View</b> .....	2-21
Key-Preserved Tables .....	2-22
Rule for DML Statements on Join Views .....	2-23
Using the UPDATABLE_COLUMNS Views.....	2-26
Outer Joins .....	2-26
<b>Managing Sequences</b> .....	2-29
Creating Sequences .....	2-29
Altering Sequences .....	2-30
Using Sequences .....	2-30
Dropping Sequences .....	2-34
<b>Managing Synonyms</b> .....	2-35
Creating Synonyms .....	2-35
Using Synonyms .....	2-35
Dropping Synonyms .....	2-36
<b>Creating Multiple Tables and Views in One Operation</b> .....	2-37
<b>Naming Schema Objects</b> .....	2-38
Name Resolution in SQL Statements .....	2-38
<b>Renaming Schema Objects</b> .....	2-39
<b>Renaming the Schema</b> .....	2-40
<b>Listing Information about Schema Objects</b> .....	2-41

### 3 Selecting a Datatype

<b>Oracle Built-In Datatypes</b> .....	3-2
Using Character Datatypes .....	3-5
Using the NUMBER Datatype .....	3-7
Using the DATE Datatype.....	3-8

Establishing Year 2000 Compliance .....	3-9
Using the LONG Datatype .....	3-18
Using RAW and LONG RAW Datatypes.....	3-20
ROWIDs and the ROWID Datatype.....	3-21
<b>ANSI/ISO, DB2, and SQL/DS Datatypes .....</b>	<b>3-24</b>
<b>Data Conversion .....</b>	<b>3-25</b>
Rule 1: Assignments .....	3-25
Rule 2: Expression Evaluation .....	3-27

## 4 Maintaining Data Integrity

<b>Using Integrity Constraints.....</b>	<b>4-2</b>
When to Enforce Business Rules with Integrity Constraints .....	4-2
When to Enforce Business Rules in Applications .....	4-3
Creating Indexes for Use with Constraints.....	4-3
Using NOT NULL Integrity Constraints .....	4-3
Setting Default Column Values .....	4-4
Choosing a Table's Primary Key .....	4-6
Using UNIQUE Key Integrity Constraints .....	4-7
<b>Using Referential Integrity Constraints.....</b>	<b>4-8</b>
Nulls and Foreign Keys .....	4-8
Relationships Between Parent and Child Tables .....	4-10
Multiple FOREIGN KEY Constraints .....	4-11
Deferring Constraint Checks.....	4-11
Managing Constraints That Have Associated Indexes .....	4-13
Concurrency Control, Indexes, and Foreign Keys.....	4-13
<b>Referential Integrity in a Distributed Database .....</b>	<b>4-14</b>
<b>Using CHECK Integrity Constraints.....</b>	<b>4-15</b>
Restrictions on CHECK Constraints .....	4-15
Designing CHECK Constraints .....	4-16
Multiple CHECK Constraints .....	4-16
CHECK and NOT NULL Integrity Constraints .....	4-16
<b>Defining Integrity Constraints .....</b>	<b>4-17</b>
The CREATE TABLE Command .....	4-17
The ALTER TABLE Command .....	4-18
Required Privileges .....	4-18

Naming Integrity Constraints .....	4-18
Why Disable Constraints? .....	4-19
Integrity Constraint Violations .....	4-20
On Definition .....	4-20
Enabling and Disabling Existing Integrity Constraints .....	4-21
Enabling and Disabling Key Integrity Constraints.....	4-22
Exception Reporting .....	4-22
<b>Altering Integrity Constraints</b> .....	4-23
Examples of MODIFY CONSTRAINT .....	4-23
<b>Dropping Integrity Constraints</b> .....	4-24
<b>Managing FOREIGN KEY Integrity Constraints</b> .....	4-25
Defining FOREIGN KEY Integrity Constraints .....	4-25
Enabling FOREIGN KEY Integrity Constraints .....	4-27
<b>Listing Integrity Constraint Definitions</b> .....	4-27
Examples.....	4-27

## 5 Selecting an Index Strategy

<b>Managing Indexes</b> .....	5-1
Creating Indexes .....	5-5
Dropping Indexes .....	5-5
<b>Function-Based Indexes</b> .....	5-6
Using Function-Based Indexes .....	5-7
Example Function-Based Indexes .....	5-11
Requirements and Restrictions for Function-Based Indexes.....	5-12
<b>Managing Clusters, Clustered Tables, and Cluster Indexes</b> .....	5-14
Guidelines for Creating Clusters.....	5-14
Performance Considerations .....	5-15
Creating Clusters, Clustered Tables, and Cluster Indexes .....	5-15
Manually Allocating Storage for a Cluster .....	5-17
Dropping Clusters, Clustered Tables, and Cluster Indexes .....	5-17
<b>Managing Hash Clusters and Clustered Tables</b> .....	5-19
Creating Hash Clusters and Clustered Tables .....	5-19
Controlling Space Usage Within a Hash Cluster .....	5-20
Dropping Hash Clusters .....	5-20
When to Use Hashing .....	5-20

<b>6</b>	<b>Speeding Up Index Access with Index-Organized Tables</b>	
	<b>Overview of Index-Organized Tables</b> .....	6-2
	Index-Organized Tables versus Ordinary Tables .....	6-2
	Advantages of Index-Organized Tables.....	6-2
	<b>Features of Index-Organized Tables</b> .....	6-4
	<b>When to Use Index-Organized Tables</b> .....	6-7
	<b>Example</b> .....	6-9
<b>7</b>	<b>Processing SQL Statements</b>	
	<b>SQL Statement Execution</b> .....	7-2
	Identifying Extensions to SQL92 (FIPS Flagging) .....	7-2
	<b>Controlling Transactions</b> .....	7-4
	Improving Performance.....	7-4
	Committing a Transaction .....	7-5
	Rolling Back a Transaction .....	7-6
	Defining a Transaction Savepoint .....	7-6
	Privileges Required for Transaction Management .....	7-7
	<b>Ensuring Repeatable Reads with Read-Only Transactions</b> .....	7-8
	<b>Using Cursors</b> .....	7-9
	Declaring and Opening Cursors .....	7-9
	Using a Cursor to Re-Execute Statements.....	7-9
	Closing Cursors .....	7-10
	Cancelling Cursors .....	7-10
	<b>Explicit Data Locking</b> .....	7-11
	Choosing a Locking Strategy .....	7-12
	Letting Oracle Control Table Locking .....	7-16
	Summary of Non-Default Locking Options .....	7-17
	Explicitly Acquiring Row Locks .....	7-18
	<b>User Locks</b> .....	7-20
	Creating User Locks .....	7-20
	Sample User Locks.....	7-20
	Viewing and Monitoring Locks.....	7-21
	<b>Concurrency Control Using Serializable Transactions</b> .....	7-22
	Serializable Transaction Interaction.....	7-25
	Setting the Isolation Level .....	7-25

Referential Integrity and Serializable Transactions.....	7-26
READ COMMITTED and SERIALIZABLE Isolation.....	7-28
Application Tips .....	7-31
<b>Autonomous Transactions</b> .....	7-32
Examples .....	7-35
Defining Autonomous Transactions.....	7-40

## 8 Dynamic SQL

<b>What Is Dynamic SQL?</b> .....	8-2
<b>When to Use Dynamic SQL</b> .....	8-3
To Execute Dynamic DML Statements.....	8-3
To Execute Statements Not Supported by Static SQL in PL/SQL .....	8-3
To Execute Dynamic Queries.....	8-4
To Reference Database Objects that Do Not Exist at Compilation.....	8-5
To Optimize Execution Dynamically .....	8-6
To Invoke Dynamic PL/SQL Blocks.....	8-7
To Perform Dynamic Operations Using Invoker-Rights.....	8-8
<b>A Dynamic SQL Scenario Using Native Dynamic SQL</b> .....	8-9
Data Model .....	8-9
Sample DML Operation.....	8-10
Sample DDL Operation .....	8-10
Sample Dynamic Single-Row Query .....	8-11
Sample Dynamic Multiple-Row Query.....	8-12
<b>Native Dynamic SQL vs. the DBMS_SQL Package</b> .....	8-12
Advantages of Native Dynamic SQL.....	8-13
Advantages of the DBMS_SQL Package .....	8-17
Examples of DBMS_SQL Package Code and Native Dynamic SQL Code .....	8-19
<b>Application Development Languages Other Than PL/SQL</b> .....	8-24

## 9 Using Procedures and Packages

<b>PL/SQL Program Units</b> .....	9-2
Anonymous Blocks .....	9-2
Stored Program Units (Procedures, Functions, and Packages) .....	9-5
<b>Wrapping PL/SQL Code</b> .....	9-28
<b>Remote Dependencies</b> .....	9-28

Timestamps.....	9-28
Signatures .....	9-30
Controlling Remote Dependencies .....	9-35
<b>Cursor Variables .....</b>	<b>9-38</b>
Declaring and Opening Cursor Variables .....	9-38
Examples of Cursor Variables.....	9-38
<b>Compile-Time Errors .....</b>	<b>9-41</b>
<b>Run-Time Error Handling .....</b>	<b>9-43</b>
Declaring Exceptions and Exception Handling Routines .....	9-44
Unhandled Exceptions .....	9-45
Handling Errors in Distributed Queries .....	9-46
Handling Errors in Remote Procedures .....	9-46
<b>Debugging Stored Procedures.....</b>	<b>9-47</b>
<b>Calling Stored Procedures.....</b>	<b>9-49</b>
<b>Calling Remote Procedures .....</b>	<b>9-53</b>
Synonyms for Procedures and Packages .....	9-56
<b>Calling Stored Functions from SQL Expressions .....</b>	<b>9-56</b>
Using PL/SQL Functions .....	9-56
Syntax .....	9-57
Naming Conventions .....	9-57
Meeting Basic Requirements.....	9-60
Controlling Side Effects .....	9-61
Overloading.....	9-69
Serially Reusable PL/SQL Packages.....	9-70

## 10 External Routines

<b>The Need to Work with Multiple Languages.....</b>	<b>10-1</b>
<b>What is an External Routine?.....</b>	<b>10-2</b>
<b>The Call Specification .....</b>	<b>10-3</b>
<b>Loading External Routines.....</b>	<b>10-4</b>
Loading Java Class Methods.....	10-4
Loading External C Routines .....	10-5
<b>Publishing an External Routine .....</b>	<b>10-6</b>
The AS LANGUAGE Clause for Java Class Methods.....	10-8
The AS LANGUAGE Clause for External C Routines .....	10-8

<b>Publishing Java Class Methods</b> .....	10-9
<b>Publishing External C Routines</b> .....	10-10
<b>Locations of Call Specifications</b> .....	10-10
<b>Passing Parameters to Java Class Methods with Call Specifications</b> .....	10-14
<b>Passing Parameters to External C Routines with Call Specifications</b> .....	10-14
Specifying Datatypes.....	10-16
External Datatype Mappings .....	10-18
BY VALUE/REFERENCE for IN and IN OUT Parameter Modes .....	10-19
The PARAMETERS Clause .....	10-20
Overriding Default Datatype Mapping.....	10-21
Specifying Properties .....	10-21
<b>Executing External Routines: the CALL Statement</b> .....	10-29
Preliminaries .....	10-30
CALL Statement Syntax.....	10-32
Calling Java Class Methods.....	10-33
Calling External C Routines.....	10-33
<b>Errors and Exceptions</b> .....	10-34
Generic Compile Time Call specification Errors.....	10-34
Java Exception Handling.....	10-35
C Exception Handling.....	10-35
<b>Using Service Routines with External C Routines</b> .....	10-35
<b>Doing Callbacks with External C Routines</b> .....	10-43
Object Support for OCI Callbacks .....	10-45
Restrictions on Callbacks.....	10-46
Debugging External Routines.....	10-47
Demo Program.....	10-48
Guidelines for External C Routines .....	10-48
Restrictions on External C Routines .....	10-50

## 11 Establishing Security Policies

<b>Introduction to Security Policies</b> .....	11-2
Security Threats and Countermeasures .....	11-2
What Discretionary Security Policies Can Cover.....	11-3
Features to Use in Establishing Security Policies.....	11-4
<b>Application Security</b> .....	11-5

Considerations for Using Application-Based Security.....	11-6
Tasks of Application Administrators .....	11-8
Overview of Roles and Application Privilege Management.....	11-8
Associating Privileges with the User's Current Application Role .....	11-9
Restricting Application Roles from Tool Users .....	11-12
Protecting Database Objects Through Use of Schemas.....	11-19
Managing Object Privileges .....	11-21
Creating a Role and Protecting Its Use .....	11-23
Enabling and Disabling Roles .....	11-24
Granting and Revoking System Privileges and Roles.....	11-28
Granting and Revoking Schema Object Privileges and Roles.....	11-30
Granting to, and Revoking from, the User Group PUBLIC .....	11-34
<b>Fine-Grained Access Control</b> .....	11-35
Features of Fine-Grained Access Control.....	11-36
How to Add a Policy to a Table or View.....	11-38
Example of a Dynamically Modified Statement .....	11-39
<b>Application Context</b> .....	11-40
Features of Application Context.....	11-41
Feature Design Principles for Application Context.....	11-45
Ways to Use Application Context with Fine Grained Access Control .....	11-47
How to Use Application Context .....	11-49
Examples .....	11-53
<b>Authentication Through a Middle Tier</b> .....	11-64
Advantages of <i>n</i> -Tier Authentication .....	11-64
Security Challenges of Three-tier Computing.....	11-65
Oracle8i <i>n</i> -Tier Authentication Solutions.....	11-69
<b>Data Encryption</b> .....	11-72
The DBMS_OBFUSCATION_TOOLKIT Package .....	11-72
Development Considerations .....	11-73

## Part III The Active Database

### 12 Using Triggers

<b>Designing Triggers</b> .....	12-2
<b>Creating Triggers</b> .....	12-3

Prerequisites for Creating Triggers .....	12-4
Types of Triggers .....	12-4
Naming Triggers .....	12-5
Triggering Statement .....	12-5
BEFORE and AFTER Options .....	12-7
INSTEAD OF Triggers .....	12-7
FOR EACH ROW Option .....	12-12
WHEN Clause .....	12-13
The Trigger Body .....	12-14
Triggers and Handling Remote Exceptions .....	12-18
Restrictions on Creating Triggers .....	12-20
Who Is the Trigger User? .....	12-26
Privileges.....	12-26
<b>Compiling Triggers</b> .....	12-28
Dependencies .....	12-28
Recompiling Triggers .....	12-29
Migration Issues .....	12-29
<b>Modifying Triggers</b> .....	12-30
Debugging Triggers .....	12-30
<b>Enabling and Disabling Triggers</b> .....	12-31
Enabling Triggers .....	12-31
Disabling Triggers .....	12-31
<b>Listing Information About Triggers</b> .....	12-32
<b>Examples of Trigger Applications</b> .....	12-34
<b>Triggering Event Publication</b> .....	12-54
Publication Framework .....	12-54

## 13 Working With System Events

<b>Event Attribute Functions</b> .....	13-2
<b>List of Events</b> .....	13-6
Resource Manager Events .....	13-6
Client Events .....	13-7

## 14 Using Publish-Subscribe

<b>Introduction to Publish-Subscribe</b> .....	14-2
--	------

<b>Publish-Subscribe Infrastructure .....</b>	<b>14-3</b>
<b>Publish-Subscribe Concepts.....</b>	<b>14-4</b>
<b>Examples.....</b>	<b>14-6</b>

## **Part IV Developing Specialized Applications**

### **15 Developing Web Applications with PL/SQL**

<b>Performing Network Operations with PL/SQL .....</b>	<b>15-1</b>
Sending Mail.....	15-1
Getting a Host Name or Address .....	15-1
Working with TCP/IP Connections.....	15-1
Retrieving the Contents of an HTTP URL.....	15-2
Working with Tables, Image Maps, Cookies, CGI Variables, and More.....	15-2
<b>Embedding PL/SQL Code in Web Pages (PL/SQL Server Pages).....</b>	<b>15-2</b>
Choosing a Software Configuration.....	15-3
Writing the Code and Content for the PL/SQL Server Page .....	15-4
Syntax of PL/SQL Server Page Elements.....	15-9
Loading the PL/SQL Server Page into the Database as a Stored Procedure.....	15-11
Running a PL/SQL Server Page via a URL .....	15-12
Examples of PL/SQL Server Pages.....	15-12
Debugging PL/SQL Server Page Problems.....	15-18
Putting an Application using PL/SQL Server Pages into Production.....	15-19

### **16 Working with Transaction Monitors with Oracle XA**

<b>X/Open Distributed Transaction Processing (DTP) .....</b>	<b>16-2</b>
Required Public Information.....	16-5
<b>XA and the Two-Phase Commit Protocol .....</b>	<b>16-5</b>
<b>Transaction Processing Monitors (TPMs) .....</b>	<b>16-6</b>
<b>Support for Dynamic and Static Registration .....</b>	<b>16-6</b>
<b>Oracle XA Library Interface Subroutines.....</b>	<b>16-7</b>
XA Library Subroutines .....	16-7
Extensions to the XA Interface .....	16-8
<b>Developing and Installing Applications That Use the XA Libraries .....</b>	<b>16-9</b>
Responsibilities of the DBA or System Administrator.....	16-9

Responsibilities of the Application Developer.....	16-10
Defining the xa_open String .....	16-10
Interfacing to Precompilers and OCIs .....	16-17
Transaction Control using XA .....	16-20
Migrating Precompiler or OCI Applications to TPM Applications .....	16-23
XA Library Thread Safety.....	16-24
<b>Troubleshooting XA Applications.....</b>	<b>16-26</b>
XA Trace Files .....	16-26
Trace File Examples.....	16-27
In-doubt or Pending Transactions .....	16-27
Oracle Server SYS Account Tables .....	16-28
<b>General XA Issues and Restrictions.....</b>	<b>16-29</b>
Database Links .....	16-29
Oracle Parallel Server Option .....	16-30
SQL-based Restrictions .....	16-30
Miscellaneous XA Issues .....	16-31
<b>Changes to Oracle XA Support .....</b>	<b>16-32</b>
XA Changes from Release 8.0 to Release 8.1 .....	16-32
XA Changes from Release 7.3 to Release 8.0 .....	16-33

## Index



---

---

# Send Us Your Comments

**Oracle8i Application Developer's Guide - Fundamentals, Release 2 (8.1.6)**

**Part No. A76939-01**

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available). You can send comments to us in the following ways:

- E-mail - [infodev@us.oracle.com](mailto:infodev@us.oracle.com)
- FAX - (650) 506-7228 Attn: Oracle Server Documentation
- Postal service:  
Oracle Corporation  
Server Documentation Manager  
500 Oracle Parkway  
Redwood Shores, CA 94065  
USA

If you would like a reply, please give your name, address, and telephone number below.

---

---

---

If you have problems with the software, please contact your local Oracle Support Services.



---

# Preface

*Application Developer's Guide - Fundamentals* describes features of application development for the Oracle Server, Release 2 (8.1.6). Information in this guide applies to versions of the Oracle Server that run on all platforms, and does not include system-specific information.

The Preface includes the following sections:

- [Information in This Guide](#)
- [Audience](#)
- [Feature Coverage and Availability](#)
- [Other Guides](#)
- [How This Book Is Organized](#)
- [Conventions Used in This Guide](#)
- [Your Comments Are Welcome](#)

## Information in This Guide

As an application developer, you should learn about the many Oracle Server features that can ease application development and improve performance. This Guide describes Oracle Server features that relate to application development. It does not cover the PL/SQL language, nor does it directly discuss application development on the client side. The table of contents and the "[How This Book Is Organized](#)" section has more information about the material covered. The "[Other Guides](#)" section points to other Oracle documentation that contains related information.

## Audience

The *Application Developer's Guide - Fundamentals* is intended for programmers developing new applications or converting existing applications to run in the Oracle environment. This Guide will also be valuable to systems analysts, project managers, and others interested in the development of database applications.

This guide assumes that you have a working knowledge of application programming, and that you are familiar with the use of Structured Query Language (SQL) to access information in relational database systems.

Certain sections of this Guide also assume a knowledge of the basic concepts of object oriented programming.

## Feature Coverage and Availability

The *Application Developer's Guide - Fundamentals* contains information that describes the features and functionality of the Oracle8i and the Oracle8i Enterprise Edition products. Oracle8i and Oracle8i Enterprise Edition have the same basic features. However, several advanced features are available only with the Enterprise Edition, and some of these are optional. For example, to use object functionality, you must have the Enterprise Edition and the Objects Option.

For information about the differences between Oracle8i and the Oracle8i Enterprise Edition and the features and options that are available to you, see *Getting to Know Oracle8i*.

## Other Guides

Use the *PL/SQL User's Guide and Reference* to learn PL/SQL and to get a complete description of this high-level programming language, which is Oracle Corporation's procedural extension to SQL.

The Oracle Call Interface (OCI) is described in *Oracle Call Interface Programmer's Guide*

You can use the OCI to build third-generation language (3GL) applications that access the Oracle Server.

Oracle Corporation also provides the Pro\* series of precompilers, which allow you to embed SQL and PL/SQL in your application programs. If you write 3GL application programs in Ada, C, C++, COBOL, or FORTRAN that incorporate embedded SQL, then refer to the corresponding precompiler manual. For example, if you program in C or C++, then refer to the *Pro\*C/C++ Precompiler Programmer's Guide*.

Oracle Developer/2000 is a cooperative development environment that provides several tools including a form builder, reporting tools, and a debugging environment for PL/SQL. If you use Developer/2000, then refer to the appropriate Oracle Tools documentation.

For SQL information, see the *Oracle8i SQL Reference* and *Oracle8i Administrator's Guide*. For basic Oracle concepts, see *Oracle8i Concepts*.

# How This Book Is Organized

The *Application Developer's Guide - Fundamentals* contains the following major sections.

## Part I: Introduction

This part introduces several different ways that you can write Oracle applications. You might need to use more than one language or development environment for a single application. Some database features are only supported, or are easier to access from, certain languages.

"[Understanding the Oracle Programmatic Environments](#)" outlines the strengths of the languages, development environments, and APIs that Oracle provides.

## Part II: Designing the Database

Before you develop an application, you need to plan the characteristics of the associated database. You must choose all the pieces that go into the database, and how they are put together. Good database design helps ensure good performance and scalability, and reduces the amount of application logic you code by making the database responsible for things like error checking and fast data access.

"[Managing Schema Objects](#)" explains how to manage objects such as tables, views, numeric sequences, and synonyms. It also discusses performance enhancements to data retrieval through the use of indexes and clusters.

"[Selecting a Datatype](#)" explains how to represent your business data in the database. The datatypes include fixed- and variable-length character strings, numeric data, dates, raw binary data, and row identifiers (ROWIDs).

"[Maintaining Data Integrity](#)" explains how to use constraints to move error-checking logic out of your application and into the database.

"[Selecting an Index Strategy](#)" and "[Speeding Up Index Access with Index-Organized Tables](#)" explain how to speed up queries.

"[Processing SQL Statements](#)" explains SQL topics such as commits, cursors, and locking that you can take advantage of in your applications.

"[Dynamic SQL](#)" describes dynamic SQL, native dynamic SQL vs. the DBMS\_SQL package, when to use dynamic SQL.

"[Using Procedures and Packages](#)" explains how to store reusable procedures in the database, and how to group procedures into packages. "[External Routines](#)" explains how to code the bodies of computationally intensive procedures in languages other than PL/SQL.

"[Establishing Security Policies](#)" explains how to move authentication logic out of your application and into the database.

### **Part III: The Active Database**

You can include all sorts of programming logic in the database itself, making the benefits available to many applications and saving repetitious coding work.

"[Using Triggers](#)" explains how to make the database do special processing before, after, or instead of running SQL statements. You can use triggers for things like validating or transforming data, or logging database access. "[Working With System Events](#)" explains how to retrieve information, such as the user ID and database name, about the event that fires a trigger.

"[Using Publish-Subscribe](#)" introduces the Oracle model for asynchronous communication, also known as messaging or queueing.

### **Part IV: Developing Specialized Applications**

"[Developing Web Applications with PL/SQL](#)" explains how to create dynamic web pages and applications that work with the Internet, e-mail, and so on, using the PL/SQL language.

"[Working with Transaction Monitors with Oracle XA](#)" describes how to connect Oracle with a transaction monitor.

## Conventions Used in This Guide

The following notational and text formatting conventions are used in this guide:

[ ]

Square brackets indicate that the enclosed item is optional. Do not type the brackets.

{ }

Braces enclose items of which only one is required.

|

A vertical bar separates items within braces, and may also be used to indicate that multiple values are passed to a function parameter.

...

In code fragments, an ellipsis means that code not relevant to the discussion has been omitted.

font change

SQL or C code examples are shown in monospaced font.

*italics*

Italics are used for OCI parameters, OCI routines names, file names, and data fields.

**UPPERCASE**

Uppercase is used for SQL keywords, like `SELECT` or `UPDATE`.

This guide uses special text formatting to draw the reader's attention to some information. A paragraph that is indented and begins with a bold text label may have special meaning. The following paragraphs describe the different types of information that are flagged this way.

**Note:** The "Note" flag indicates that you should pay particular attention to the information to avoid a common problem or to increase understanding of a concept.

**Warning:** An item marked as "Warning" indicates something that an OCI programmer must be careful to do, or not do, in order for an application to work correctly.

**See Also:** Text marked "See Also" points you to another section of this guide, or to other documentation, for additional information about the topic being discussed.

## Your Comments Are Welcome

We value and appreciate your comment as an Oracle user and reader of our manuals. As we write, revise, and evaluate our documentation, your opinions are the most important feedback we receive.

You can send comments and suggestions about this manual to the following e-mail address:

[infodev@us.oracle.com](mailto:infodev@us.oracle.com)

If you prefer, then you can send letters or faxes containing your comments to the Information Development department at the following address:

Server Technologies Documentation Manager

Oracle Corporation

500 Oracle Parkway

Redwood Shores, CA 94065

Fax: (650) 506-7200



# Part I

---

## Introduction To Working With The Server

This part contains the following chapter:

- [Chapter 1, "Understanding the Oracle Programmatic Environments"](#)



---

# Understanding the Oracle Programmatic Environments

This chapter presents brief introductions to these application development systems:

- Overview of PL/SQL
- Overview of OCI
- Overview of Oracle Objects for OLE
- Overview of Pro\*C/C++
- Overview of Pro\*COBOL
- Overview of Oracle JDBC
- Overview of Oracle SQLJ
- Choosing a Programming Environment

## Overview of PL/SQL

PL/SQL is Oracle's procedural extension to SQL, the standard database access language. An advanced 4GL (fourth-generation programming language), PL/SQL offers seamless SQL access, tight integration with the Oracle server and tools, portability, security, and modern software engineering features such as data encapsulation, overloading, exception handling, and information hiding.

With PL/SQL, you can use SQL statements to manipulate Oracle data and flow-of-control statements to process the data. You can also declare constants and variables, define procedures and functions, use collections and object types, and trap run-time errors. Thus, PL/SQL combines the data manipulating power of SQL with the data processing power of procedural languages.

Applications written using any of the Oracle programmatic interfaces (Oracle Call Interface, Java, Pro\*C/C++, or Pro\*COBOL) can call PL/SQL stored procedures and send anonymous blocks of PL/SQL code to the server for execution. 3GL applications have full access to PL/SQL scalar and composite datatypes via host variables and implicit datatype conversion.

PL/SQL's tight integration with Oracle Developer lets you use one language to develop the client and server components of your application, which can be partitioned to achieve optimum performance and scalability. Also, Oracle's Web Forms allows you to deploy your applications in a multi-tier Internet or intranet environment without modifying a single line of code.

For more information see *PL/SQL User's Guide and Reference*.

## How Does PL/SQL Work?

A good way to get acquainted with PL/SQL is to look at a sample program. Consider the procedure below, which debits a bank account. When called, procedure `debit_account` accepts an account number and a debit amount. It uses the account number to select the account balance from the database table. Then, it uses the debit amount to compute a new balance. If the new balance is less than zero, an exception is raised; otherwise, the bank account is updated.

```
PROCEDURE debit_account (acct_id INTEGER, amount REAL) IS
    old_balance REAL;
    new_balance REAL;
    overdrawn    EXCEPTION;
BEGIN
    SELECT bal INTO old_balance FROM accts
        WHERE acct_no = acct_id;
    new_balance := old_balance - amount;
```

```
IF new_balance < 0 THEN
    RAISE overdrawn;
ELSE
    UPDATE accts SET bal = new_balance
        WHERE acct_no = acct_id;
END IF;
COMMIT;
EXCEPTION
    WHEN overdrawn THEN
        -- handle the error
END debit_account;
```

## Advantages of PL/SQL

PL/SQL is a completely portable, high-performance transaction processing language that offers the following advantages:

### Full Support for SQL

PL/SQL lets you use all the SQL data manipulation, cursor control, and transaction control commands, as well as all the SQL functions, operators, and pseudocolumns. So, you can manipulate Oracle data flexibly and safely. Also, PL/SQL fully supports SQL datatypes. That reduces the need to convert data passed between your applications and the database.

PL/SQL also supports dynamic SQL, an advanced programming technique that makes your applications more flexible and versatile. Your programs can build and process SQL data definition, data control, and session control statements "on the fly" at run time.

### Tight Integration with Oracle

Both PL/SQL and Oracle are based on SQL, and PL/SQL supports all the SQL datatypes. Combined with the direct access that SQL provides, these shared datatypes integrate PL/SQL with the Oracle data dictionary.

The %TYPE and %ROWTYPE attributes further integrate PL/SQL with the data dictionary. For example, you can use the %TYPE attribute to declare variables, basing the declarations on the definitions of database columns. If a definition changes, the variable declaration changes accordingly at run time. This provides data independence, reduces maintenance costs, and allows programs to adapt as the database changes to meet new business needs.

### **Better Performance**

Without PL/SQL, Oracle must process SQL statements one at a time. Each SQL statement results in another call to Oracle and higher performance overhead. In a networked environment, the overhead can become significant. Every time a SQL statement is issued, it must be sent over the network, creating more traffic.

However, with PL/SQL, an entire block of statements can be sent to Oracle at one time. This can drastically reduce communication between your application and Oracle. If your application is database intensive, you can use PL/SQL blocks to group SQL statements before sending them to Oracle for execution.

PL/SQL stored procedures are compiled once and stored in executable form, so procedure calls are quick and efficient. Also, stored procedures, which execute in the server, can be invoked over slow network connections with a single call. This reduces network traffic and improves round-trip response times. Executable code is automatically cached and shared among users. That lowers memory requirements and invocation overhead.

### **Higher Productivity**

PL/SQL adds capabilities to non-procedural tools such as Oracle Forms and Oracle Reports. With PL/SQL in these tools, you can use familiar procedural constructs to build applications. For example, you can use an entire PL/SQL block in an Oracle Forms trigger. You need not use multiple trigger steps, macros, or user exits. Thus, PL/SQL increases productivity by putting better tools in your hands.

PL/SQL is the same in all environments. As soon as you master PL/SQL with one Oracle tool, you can transfer your knowledge to other tools, and so multiply the productivity gains. For example, scripts written with one tool can be used by other tools.

### **Scalability**

PL/SQL stored procedures increase scalability by isolating application processing on the server. Also, automatic dependency tracking for stored procedures aids the development of scalable applications.

The shared memory facilities of the Multithreaded Server (MTS) enable Oracle to support 10,000+ concurrent users on a single node. For more scalability, you can use the Net8 Connection Manager to multiplex Net8 connections.

## Maintainability

Once validated, a PL/SQL stored procedure can be used with confidence in any number of applications. If its definition changes, only the procedure is affected, not the applications that call it. This simplifies maintenance and enhancement. Also, maintaining a procedure on the server is easier than maintaining copies on various client machines.

## Support for Object-Oriented Programming

**Object Types** An object type is a user-defined composite datatype that encapsulates a data structure along with the functions and procedures needed to manipulate the data. The variables that form the data structure are called attributes. The functions and procedures that characterize the behavior of the object type are called methods, which you can implement in PL/SQL.

Object types are an ideal object-oriented modeling tool, which you can use to reduce the cost and time required to build complex applications. Besides allowing you to create software components that are modular, maintainable, and reusable, object types allow different teams of programmers to develop software components concurrently.

**Collections** A collection is an ordered group of elements, all of the same type (for example, the grades for a class of students). Each element has a unique subscript that determines its position in the collection. PL/SQL offers two kinds of collections: nested tables and VARRAYs (short for variable-size arrays).

Collections, which work like the arrays found in most third-generation programming languages, can store instances of an object type and, conversely, can be attributes of an object type. Also, collections can be passed as parameters. So, you can use them to move columns of data into and out of database tables or between client-side applications and stored subprograms. Furthermore, you can define collection types in a PL/SQL package, then use them programmatically in your applications.

## Portability

Applications written in PL/SQL can run on any operating system and hardware platform where Oracle runs. You can write portable program libraries and reuse them in different environments.

## **Security**

PL/SQL stored procedures enable you to partition application logic between the client and server. That way, you can prevent client applications from manipulating sensitive Oracle data. Database triggers written in PL/SQL can disable application updates selectively and do content-based auditing of user queries.

Furthermore, you can restrict access to Oracle data by allowing users to manipulate it only through stored procedures that execute with their definer's privileges. For example, you can grant users access to a procedure that updates a table, but not grant them access to the table itself.

## Overview of OCI

The Oracle Call Interface (OCI) is an application programming interface (API) that allows you to create applications that use a third-generation language's native procedures or function calls to access an Oracle database server and control all phases of SQL statement execution. OCI provides:

- Improved performance and scalability through the efficient use of system memory and network connectivity
- Consistent interfaces for dynamic session and transaction management in a two-tier client-server or multi-tier environment
- N-tiered authentication
- Comprehensive support for application development using Oracle objects
- Access to external databases
- Ability to develop applications that service an increasing number of users and requests without additional hardware investments

OCI allows you to manipulate data and schemas in an Oracle database using a host programming language, such as C. It provides a library of standard database access and retrieval functions in the form of a dynamic runtime library (OCILIB) that can be linked in an application at runtime. This eliminates the need to embed SQL or PL/SQL within 3GL programs.

For more information about the OCI calls, see *Oracle Call Interface Programmer's Guide*, *Oracle8i Application Developer's Guide - Advanced Queuing*, *Oracle8i National Language Support Guide*, and *Oracle8i Data Cartridge Developer's Guide*.

## Advantages of OCI

OCI provides significant advantages over other methods of accessing an Oracle database:

- More fine-grained control over all aspects of the application design.
- High degree of control over program execution.
- Use of familiar 3GL programming techniques and application development tools such as browsers and debuggers.
- Support of dynamic SQL, method 4.
- Availability on the broadest range of platforms of all the Oracle programmatic interfaces.

- Dynamic bind and define using callbacks.
- Describe functionality to expose layers of server metadata.
- Asynchronous event notification for registered client applications.
- Enhanced array data manipulation language (DML) capability for array `INSERTs`, `UPDATEs`, and `DELETEs`.
- Ability to associate a commit request with an execute to reduce roundtrips.
- Optimization for queries using transparent prefetch buffers to reduce roundtrips.
- Thread safety so you do not have to use mutual exclusive locks (mutex) on OCI handles.
- The server connection in non-blocking mode means that control returns to the OCI code when a call is still executing or could not complete.

## Parts of the OCI

The OCI encompasses four main sets of functionality:

- OCI *relational functions*, for managing database access and processing SQL statements
- OCI *navigational functions*, for manipulating objects retrieved from an Oracle database server
- OCI *datatype mapping and manipulation functions*, for manipulating data attributes of Oracle types
- OCI *external procedure functions*, for writing C callbacks from PL/SQL

## Procedural and Non-Procedural Elements

The Oracle Call Interface (OCI) allows you to develop applications that combine the non-procedural data access power of Structured Query Language (SQL) with the procedural capabilities of most programming languages, such as C and C++.

- In a non-procedural language program, the set of data to be operated on is specified, but what operations will be performed, or how the operations are to be carried out is not specified. The non-procedural nature of SQL makes it an easy language to learn and to use to perform database transactions. It is also the standard language used to access and manipulate data in modern relational and object-relational database systems.

- In a procedural language program, the execution of most statements depends on previous or subsequent statements and on control structures, such as loops or conditional branches, which are not available in SQL. The procedural nature of these languages makes them more complex than SQL, but it also makes them very flexible and powerful.

The combination of both non-procedural and procedural language elements in an OCI program provides easy access to an Oracle database in a structured programming environment.

The OCI supports all SQL data definition, data manipulation, query, and transaction control facilities that are available through an Oracle database server. For example, an OCI program can run a query against an Oracle database. The queries can require the program to supply data to the database using input (bind) variables, as follows:

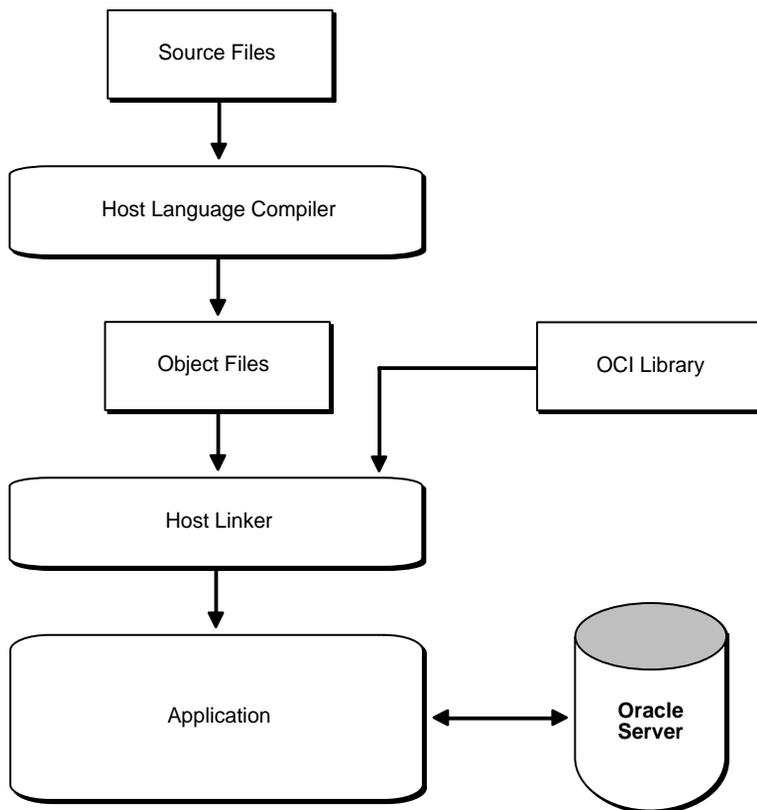
```
SELECT name FROM employees WHERE empno = :empnumber
```

In the above SQL statement, `:empnumber` is a placeholder for a value that will be supplied by the application.

You can also use PL/SQL, Oracle's procedural extension to SQL. The applications you develop can be more powerful and flexible than applications written in SQL alone. The OCI also provides facilities for accessing and manipulating objects in an Oracle database server.

## Building an OCI Application

As [Figure 1-1](#) shows, you compile and link an OCI program in the same way that you compile and link a non-database application. There is no need for a separate preprocessing or precompilation step.

**Figure 1-1 The OCI Development Process**

**Note:** On some platforms, it may be necessary to include other libraries, in addition to the OCI library, to properly link your OCI programs. Check your Oracle system-specific documentation for further information about extra libraries that may be required.

## Overview of Oracle Objects for OLE

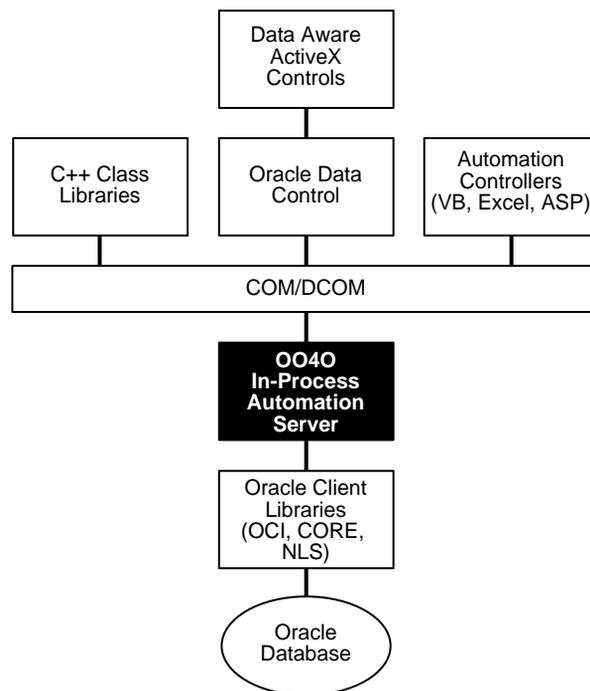
Oracle Objects for OLE (OO4O) is a product designed to allow easy access to data stored in Oracle databases with any programming or scripting language that supports the Microsoft COM Automation and ActiveX technology. This includes Visual Basic, Visual C++, Visual Basic For Applications (VBA), IIS Active Server Pages (VBScript and JavaScript), and others.

OO4O consists of the following software layers:

- OO4O "In-Process" Automation Server
- Oracle Data Control
- Oracle Objects for OLE C++ Class Library

Figure 1-2, "Software Layers" illustrates the OO4O software components.

**Figure 1-2 Software Layers**



**Note:** See the OO4O online help for detailed information about using OO4O.

## The OO4O Automation Server

The OO4O Automation Server is a set of COM Automation objects for connecting to Oracle database servers, executing SQL statements and PL/SQL blocks, and accessing the results.

Unlike other COM-based database connectivity APIs, such as Microsoft ADO, the OO4O Automation Server has been developed and evolved specifically for use with Oracle database servers.

It provides an optimized API for accessing features that are unique to Oracle and are otherwise cumbersome or inefficient to use from ODBC or OLE database-specific components.

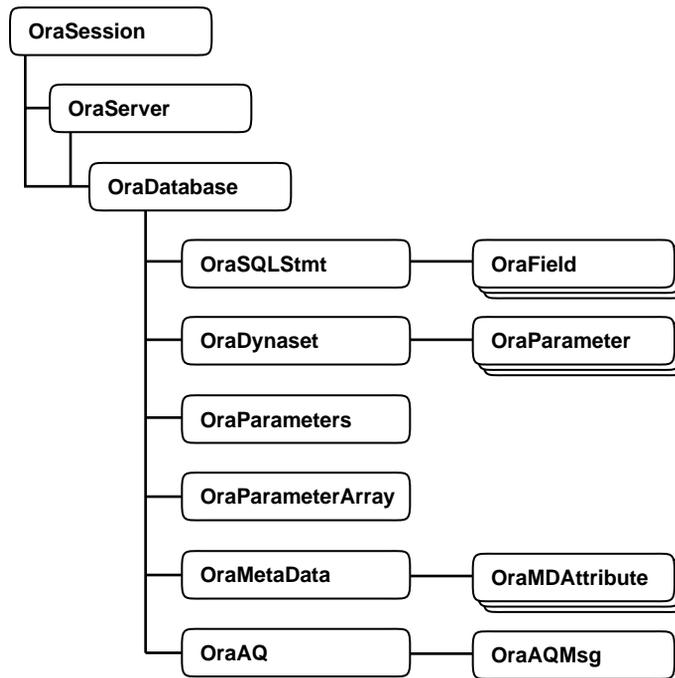
OO4O provides key features for accessing Oracle databases efficiently and easily in environments ranging from the typical two-tier client/server applications, such as those developed in Visual Basic or Excel, to application servers deployed in multi-tiered application server environments such as web server applications in Microsoft Internet Information Server (IIS) or Microsoft Transaction Server (MTS).

Features include:

- Support for execution of PL/SQL anonymous blocks and stored procedures. This includes support for Oracle datatypes allowed for input/output parameters of PL/SQL stored procedures including PL/SQL cursors. See ["Support for Oracle LOB and Object Datatypes"](#) on page 1-17.
- Support for scrollable and updateable cursors for easy and efficient access to result sets of queries.
- Thread-safe objects and Connection Pool Management Facility for developing efficient web server applications.
- Full support for Oracle8i Object-Relational and LOB datatypes.
- Full support for Advanced Queuing in Oracle8i
- Support for array inserts and updates.
- Support for Microsoft Transaction Server (MTS).

## OO4O Object Model

The Oracle Objects for OLE object model is illustrated in [Figure 1-3, "Objects and Their Relation"](#).

**Figure 1–3 Objects and Their Relation**

### OraSession

An OraSession object manages collections of OraDatabase, OraConnection, and OraDynaset objects used within an application.

Typically, a single OraSession object is created for each application, but you can create named OraSession objects for shared use within and between applications.

The OraSession object is the top-most level object for an application. It is the only object created by the CreateObject VB/VBA API and not by an Oracle Objects for OLE method. The following code fragment shows how to create an OraSession object:

```
Dim OraSession as Object
Set OraSession = CreateObject("OracleInProcServer.XOraSession")
```

### OraServer

OraServer represents a physical network connection to an Oracle database server.

The OraServer interface is introduced to expose the connection multiplexing feature provided in the Oracle Call Interface. After an OraServer object is created, multiple user sessions (OraDatabase) can be attached to it by invoking the OpenDatabase method. This feature is particularly useful for application components, such as Internet Information Server (IIS), that use Oracle Objects for OLE in an n-tier distributed environments.

The use of connection multiplexing when accessing Oracle servers with a large number of user sessions active can help reduce server processing and resource requirements while improving the server scalability.

### **OraDatabase**

An OraDatabase interface in the Oracle8*i* release adds additional methods for controlling transactions and creating interfaces representing of Oracle object types. Attributes of schema objects can be retrieved using the Describe method of the OraDatabase interface.

In previous releases, an OraDatabase object is created by invoking the OpenDatabase method of an OraSession interface. The Net8 alias, user name, and password are passed as arguments to this method. In the Oracle8*i* release, invocation of this method results in implicit creation of an OraServer object.

As described in the OraServer interface description, an OraDatabase object can also be created using the OpenDatabase method of the OraServer interface.

Transaction control methods are available at the OraDatabase (user session) level. Transactions may be started as Read-Write (default), Serializable, or Read-only. These include:

- BeginTrans
- CommitTrans
- RollbackTrans

For example:

```
UserSession.BeginTrans(OO4O_TXN_READ_WRITE)
UserSession.ExecuteSQL("delete emp where empno = 1234")
UserSession.CommitTrans
```

### **OraDynaset**

An OraDynaset object permits browsing and updating of data created from a SQL SELECT statement.

The OraDynaset object can be thought of as a cursor, although in actuality several real cursors may be used to implement the OraDynaset's semantics. An OraDynaset automatically maintains a local cache of data fetched from the server and transparently implements scrollable cursors within the browse data. Large queries may require significant local disk space; application implementers are encouraged to refine queries to limit disk usage.

### **OraField**

An OraField object represents a single column or data item within a row of a dynaset.

If the current row is being updated, then the OraField object represents the currently updated value, although the value may not yet have been committed to the database.

Assignment to the Value property of a field is permitted only if a record is being edited (using Edit) or a new record is being added (using AddNew). Other attempts to assign data to a field's Value property results in an error.

### **OraMetaData**

An OraMetaData object is a collection of OraMDAttribute objects that represent the description information about a particular schema object in the database.

The OraMetaData object can be visualized as a table with three columns:

- Metadata Attribute Name
- Metadata Attribute Value
- Flag specifying whether the Value is another OraMetaData Object

The OraMDAttribute objects contained in the OraMetaData object can be accessed by subscripting using ordinal integers or by using the name of the property. Referencing a subscript that is not in the collection (0 to Count-1) results in the return of a NULL OraMDAttribute object.

### **OraParameter**

An OraParameter object represents a bind variable in a SQL statement or PL/SQL block.

OraParameter objects are created, accessed, and removed indirectly through the OraParameters collection of an OraDatabase object. Each parameter has an identifying name and an associated value. You can automatically bind a parameter

to SQL and PL/SQL statements of other objects (as noted in the objects' descriptions), by using the parameter's name as a placeholder in the SQL or PL/SQL statement. Such use of parameters can simplify dynamic queries and increase program performance.

### **OraParamArray**

An OraParamArray object represents an "array" type bind variable in a SQL statement or PL/SQL block as opposed to a "scalar" type bind variable represented by the OraParameter object.

OraParamArray objects are created, accessed, and removed indirectly through the OraParameters collection of an OraDatabase object. Each parameter has an identifying name and an associated value.

### **OraSQLStmt**

An OraSQLStmt Object represents a single SQL statement. Use the CreateSQL method to create the OraSQLStmt object from an OraDatabase.

During create and refresh, OraSQLStmt objects automatically bind all relevant, enabled input parameters to the specified SQL statement, using the parameter names as placeholders in the SQL statement. This can improve the performance of SQL statement execution without re-parsing the SQL statement.

### **SQLStmt**

The SQLStmt object (updateStmt) can be later used to execute the same query using a different value for the :SALARY placeholder. This is done as follows:

```
OraDatabase.Parameters("SALARY").value = 200000  
updateStmt.Parameters("ENAME").value = "KING"  
updateStmt.Refresh
```

### **OraAQ**

An OraAQ object is instantiated by invoking the CreateAQ method of the OraDatabase interface. It represents a queue that is present in the database.

Oracle Objects for OLE provides interfaces for accessing Oracle's Advanced Queuing (AQ) Feature. It makes AQ accessible from popular COM-based development environments such as Visual Basic. For a detailed description of Oracle AQ, please refer to *Oracle8i Application Developer's Guide - Advanced Queuing*.

### **OraAQMsg**

The OraAQMsg object encapsulates the message to be enqueued or dequeued. The message can be of any user-defined or raw type.

For a detailed description of Oracle AQ, please refer to *Oracle8i Application Developer's Guide - Advanced Queuing*.

### **OraAQAgent**

The OraAQAgent object represents a message recipient and is only valid for queues which allow multiple consumers.

The OraAQAgent object represents a message recipient and is only valid for queues which allow multiple consumers.

An OraAQAgent object can be instantiated by invoking the AQAgent method. For example:

```
Set agent = qMsg.AQAgent(name)
```

An OraAQAgent object can also be instantiated by invoking the AddRecipient method. For example:

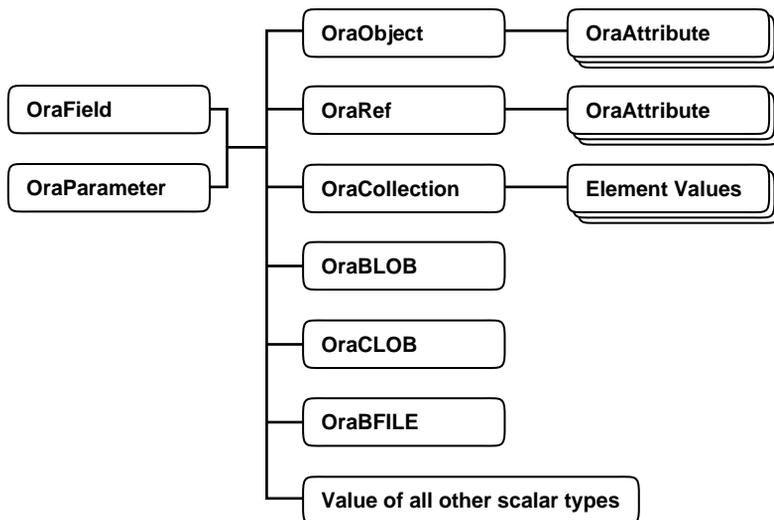
```
Set agent = qMsg.AddRecipient(name, address, protocol).
```

## **Support for Oracle LOB and Object Datatypes**

Oracle Objects for OLE provides full support for accessing and manipulating instances of object datatypes and LOBs in an Oracle database server. [Figure 1-4, "Supported Oracle Datatypes"](#) illustrates the datatypes supported by OO4O.

Instances of these types can be fetched from the database or passed as input or output variables to SQL statements and PL/SQL blocks, including stored procedures and functions. All instances are mapped to COM Automation Interfaces that provide methods for dynamic attribute access and manipulation. These interfaces may be obtained from:

**Figure 1–4 Supported Oracle Datatypes**



### **OraBLOB and OraCLOB**

The OraBlob and OraClob interfaces in OO4O provide methods for performing operations on large objects in the database of data types BLOB, CLOB, and NCLOB. In this help file BLOB, CLOB, and NCLOB datatypes are also referred to as LOB datatypes.

LOB data is accessed using Read and the CopyToFile methods.

LOB data is modified using Write, Append, Erase, Trim, Copy, CopyFromFile, and CopyFromBFile methods. Before modifying the content of a LOB column in a row, a row lock must be obtained. If the LOB column is a field of an OraDynaset, then the lock is obtained by invoking the Edit method.

### **OraBFILE**

The OraBFile interface in OO4O provides methods for performing operations on large objects BFILE data type in the database.

The BFILES are large binary data objects stored in operating system files (external) outside of the database tablespaces.

## The Oracle Data Control

The Oracle Data Control (ODC) is an ActiveX Control that is designed to simplify the exchange of data between an Oracle database and visual controls such edit, text, list, and grid controls in Visual Basic and other development tools that support custom controls.

ODC acts an agent to handle the flow of information from an Oracle database and a visual data-aware control, such as a grid control, that is bound to it. The data control manages various user interface (UI) tasks such as displaying and editing data. It also executes and manages the results of database queries.

The Oracle Data Control is compatible with the Microsoft data control included with Visual Basic. If you are familiar with the Visual Basic data control, learning to use the Oracle Data Control is quick and easy. Communication between data-aware controls and a Data Control is governed by a protocol that has been specified by Microsoft.

## The Oracle Objects for OLE C++ Class Library

The Oracle Objects for OLE C++ Class Library is a collection of C++ classes that provide programmatic access to the Oracle Object Server. Although the class library is implemented using OLE Automation, neither the OLE development kit nor any OLE development knowledge is necessary to use it. This library helps C++ developers avoid the chore of writing COM client code for accessing the OO4O interfaces.

## Additional Sources of Information

For detailed information about Oracle Objects for OLE refer to the online help that is provided with the OO4O product:

- Oracle Objects for OLE Help
- Oracle Objects for OLE C++ Class Library Help

To view examples of the use of Oracle Object for OLE, see the samples located in the ORACLE\_HOME\OO4O directory of the Oracle installation. Additional OO4O examples can be found in the following Oracle publications, including:

- *Oracle8i Application Developer's Guide - Large Objects (LOBs)*
- *Oracle8i Application Developer's Guide - Advanced Queuing*
- *Oracle8i Supplied PL/SQL Packages Reference*

## Overview of Pro\*C/C++

The Pro\*C/C++ precompiler is a software tool that allows the programmer to embed SQL statements in a C or C++ source file. Pro\*C/C++ reads the source file as input and outputs a C or C++ source file that replaces the embedded SQL statements with Oracle runtime library calls, and is then compiled by the C or C++ compiler.

When there are errors found during the precompilation or the subsequent compilation, modify your precompiler input file and re-run the two steps.

## How You Implement a Pro\*C/C++ Application

Here is a simple code fragment from a C source file that queries the table EMP which is in the schema SCOTT:

```
...
#define UNAME_LEN 10
...
int emp_number;
/* Define a host structure for the output values of a SELECT statement. */
/* No declare section needed if precompiler option MODE=ORACLE */
struct {
    VARCHAR emp_name[UNAME_LEN];
    float salary;
    float commission;
} emprec;
/* Define an indicator structure to correspond to the host output structure. */
struct {
    short emp_name_ind;
    short sal_ind;
    short comm_ind;
} emprec_ind;
...
/* Select columns ename, sal, and comm given the user's input for empno. */
EXEC SQL SELECT ename, sal, comm
        INTO :emprec INDICATOR :emprec_ind
        FROM emp
        WHERE empno = :emp_number;
...
```

The embedded SELECT statement is only slightly different from an interactive (SQL\*Plus) version. Every embedded SQL statement begins with EXEC SQL. The colon, ':', precedes every host (C) variable. The returned values of data and indicators (set when the data value is NULL or character columns have been

truncated) can be stored in structs (such as in the above code fragment), in arrays, or in arrays of structs. Multiple result set values are handled very simply in a manner that resembles the case shown, where there is only one result, because of the unique employee number. You use the actual names of columns and tables in embedded SQL.

Use the default precompiler option values, or you can enter values which give you control over the use of resources, how errors are reported, the formatting of output, and how cursors (which correspond to a particular connection, a SQL statement, etc.) are managed. Cursors are used when there are multiple result set values.

Enter the options either in a configuration file, on the command line, or inline inside your source code with a special statement that begins with EXEC ORACLE. If there are no errors found, you can then compile, link, and execute the output source file, like any other C program that you write.

Use the precompiler to create server database access from clients that can be on many different platforms. Pro\*C/C++ allows you the freedom to design your own user interfaces and to add database access to existing applications.

Before writing your embedded SQL statements, you may want to test interactive versions of the SQL in SQL\*Plus. You then make only minor changes to start testing your embedded SQL application.

## Highlights of Pro\*C/C++ Features

The following is a short subset of the capabilities of Pro\*C/C++. For complete details, see the *Pro\*C/C++ Precompiler Programmer's Guide*.

You can write your application in either C or C++.

You can write multi-threaded programs if your platform supports a threads package. Concurrent connections are supported in either single-threaded or multi-threaded applications.

You can improve performance by embedding PL/SQL blocks. These blocks can call functions or procedures written by you or provided in Oracle packages, in either Java or PL/SQL.

Using precompiler options, you can check the syntax and semantics of your SQL or PL/SQL statements during precompilation, as well as at runtime.

You can call stored PL/SQL and Java subprograms. Modules written in COBOL or in C can be called from Pro\*C/C++. External C procedures in shared libraries are callable by your program.

You can conditionally precompile sections of your code so that they can execute in different environments.

You can use arrays, or structures, or arrays of structures as host and indicator variables in your code to improve performance.

You can deal with errors and warnings so that data integrity is guaranteed. As a programmer, you control how errors are handled.

Your program can convert between internal datatypes and C language datatypes.

The Oracle Call Interface (OCI), a lower-level C interface, is available for use in your precompiler source.

Pro\*C/C++ supports dynamic SQL, a technique that allows users to input variable values and statement syntax.

## **New Oracle8i Features Supported**

Pro\*C/C++ can use special SQL statements to manipulate tables containing user-defined object types. An Object Type Translator (OTT) will map the object types and named collection types in your database to structures and headers that you will then include in your source.

Two kinds of collection types, nested tables and VARRAYs, are supported with a set of SQL statements that allow a high degree of control over data.

Large Objects (LOBs, CLOBs, NCLOBs, and external files known as BFILEs) are accessed by another set of SQL statements.

A new ANSI SQL standard for dynamic SQL is supported for new applications, so that you can execute SQL statements with a varying number of host variables. An older technique for dynamic SQL is still usable by pre-existing applications.

National Language Support for multi-byte characters and UCS2 Unicode support are provided.

## Overview of Pro\*COBOL

The Pro\*COBOL precompiler is a software tool that allows the programmer to embed SQL statements in a COBOL source code file. Pro\*COBOL reads the source file as input and outputs a COBOL source file that replaces the embedded SQL statements with Oracle runtime library calls, and is then compiled by the COBOL compiler.

When there are errors found during the precompilation or the subsequent compilation, modify your precompiler input file and re-run the two steps.

## How You Implement a Pro\*COBOL Application

Here is a simple code fragment from a source file that queries the table EMP which is in the schema SCOTT:

```

...
WORKING-STORAGE SECTION.
*
* DEFINE HOST INPUT AND OUTPUT HOST AND INDICATOR VARIABLES.
* NO DECLARE SECTION NEEDED IF MODE=ORACLE.
*
01 EMP-REC-VARS.
   05 EMP-NAME      PIC X(10) VARYING.
   05 EMP-NUMBER    PIC S9(4) COMP VALUE ZERO.
   05 SALARY        PIC S9(5)V99 COMP-3 VALUE ZERO.
   05 COMMISSION    PIC S9(5)V99 COMP-3 VALUE ZERO.
   05 COMM-IND      PIC S9(4) COMP VALUE ZERO.
...
PROCEDURE DIVISION.
...
EXEC SQL
    SELECT ENAME, SAL, COMM
    INTO :EMP-NAME, :SALARY, :COMMISSION:COMM-IND
    FROM EMP
    WHERE EMPNO = :EMP_NUMBE
END-EXEC.
...

```

The embedded `SELECT` statement is only slightly different from an interactive (SQL\*Plus) version. Every embedded SQL statement begins with `EXEC SQL`. The colon, `'`, precedes every host (COBOL) variable. The SQL statement is terminated by `END-EXEC`. The returned values of data and indicators (set when the data value is NULL or character columns have been truncated) can be stored in group items

(such as in the above code fragment), in tables, or in tables of group items. Multiple result set values are handled very simply in a manner that resembles the case shown, where there is only one result, given the unique employee number. You use the actual names of columns and tables in embedded SQL.

Use the default precompiler option values, or you can enter values which give you control over the use of resources, how errors are reported, the formatting of output, and how cursors (which correspond to a particular connection, a SQL statement, etc.) are managed.

Enter the options either in a configuration file, on the command line, or inline inside your source code with a special statement that begins with EXEC ORACLE. If there are no errors found, you can then compile, link, and execute the output source file, like any other COBOL program that you write.

Use the precompiler to create server database access from clients that can be on many different platforms. Pro\*COBOL allows you the freedom to design your own user interfaces and to add database access to existing COBOL applications.

The embedded SQL statements available conform to an ANSI standard, so that you can access data from many databases in a program, including remote servers, networked through Net8.

Before writing your embedded SQL statements, you may want to test interactive versions of the SQL in SQL\*Plus. You then make only minor changes to start testing your embedded SQL application.

## Highlights of Pro\*COBOL Features

The following is a short subset of the capabilities of Pro\*COBOL. For complete details, see the *Pro\*COBOL Precompiler Programmer's Guide*.

You can call stored PL/SQL or Java subprograms. You can improve performance by embedding PL/SQL blocks. These blocks can call PL/SQL functions or procedures written by you or provided in Oracle packages.

Precompiler options allow you to define how cursors, errors, syntax-checking, file formats, etc., are handled.

Using precompiler options, you can check the syntax and semantics of your SQL or PL/SQL statements during precompilation, as well as at runtime.

You can conditionally precompile sections of your code so that they can execute in different environments.

Use tables, or group items, or tables of group items as host and indicator variables in your code to improve performance.

You can program how errors and warnings are handled, so that data integrity is guaranteed.

Pro\*COBOL supports dynamic SQL, a technique that allows users to input variable values and statement syntax.

## **New Oracle8i Features Supported**

Large Objects (LOBs, CLOBs, NCLOBs, and external files known as BFILEs) are accessed by another set of SQL statements.

A new ANSI SQL standard for dynamic SQL is supported for new applications, so that you can execute SQL statements with a varying number of host variables. An older technique for dynamic SQL is still usable by pre-existing applications.

Pro\*COBOL has many features that are compatible with DB2, for easier migration.

## Overview of Oracle JDBC

JDBC (Java Database Connectivity) is an API (Applications Programming Interface) which allows Java to send SQL statements to an object-relational database such as Oracle8i.

The JDBC standard defines four types of JDBC drivers:

- Type 1. A JDBC-ODBC bridge. Software must be installed on client systems.
- Type 2. Has Native methods (calls C or C++) and Java methods. Software must be installed on the client.
- Type 3. Pure Java. The client uses sockets to call middleware on the server.
- Type 4. The most pure Java solution. Talks directly to the database using Java sockets.

JDBC is based on the X/Open SQL Call Level Interface, and complies with the SQL92 Entry Level standard.

Use JDBC to do dynamic SQL. Dynamic SQL means that the embedded SQL statement to be executed is not known before the application is run, and requires input to build the statement.

The drivers that are implemented by Oracle have extensions to the capabilities in the JDBC standard that was defined by Sun Microsystems. Oracle's implementations of JDBC drivers are described next:

### JDBC Thin Driver

The JDBC Thin driver is a Type 4 (100% pure Java) driver that uses Java sockets to connect directly to a database server. It has its own implementation of a TTC, a lightweight implementation of a TCP/IP version of Oracle's Net8. It is written entirely in Java and is therefore platform-independent.

The Thin driver does not require Oracle software on the client side. It does need a TCP/IP listener on the server side. Use this driver in Java applets that are downloaded into a Web browser. The Thin driver is self-contained, but it opens a Java socket, and thus can only run in a browser that supports sockets.

### JDBC OCI Driver

The OCI driver is a Type 2 JDBC driver. It makes calls to the OCI (Oracle Call Interface) which is written in C, to interact with an Oracle database server, thus using native and Java methods.

Because it uses native methods (a combination of Java and C) the OCI driver is platform-specific. It requires a client Oracle8i installation including Net8, OCI libraries, CORE libraries, and all other dependent files. The OCI driver usually executes faster than the thin driver.

The OCI driver is not appropriate for Java applets, because it uses a C library that is platform-specific and cannot be downloaded into a Web browser. It is usable in the Oracle Web Application Server which is a collection of middleware services and tools that supports access from and to applications from browsers and CORBA (Common Object Request Broker Architecture) clients.

## The JDBC Server Driver

The JDBC server driver is a Type 2 driver that runs inside the database server and therefore reduces the number of round-trips needed to access large amounts of data. The driver, the Java server VM, the database, the NCOMP native compiler which speeds execution by as much as 10 times, and the SQL engine all run within the same address space.

This driver provides server-side support for any Java program used in the database: SQLJ stored procedures, functions, and triggers, Java stored procedures, CORBA objects, and EJB (Enterprise Java Beans). You can also call PL/SQL stored procedures, functions, and triggers.

The server driver fully supports the same features and extensions as the client-side drivers.

## Extensions of JDBC

Among the Oracle extensions to the JDBC 1.22 standard are:

- Support for Oracle datatypes
- Performance enhancement by row prefetching
- Performance enhancement by execution batching
- Specification of query column types to save round-trips
- Control of DatabaseMetaData calls

## Sample Program for the JDBC Thin Driver

The following source code registers an Oracle JDBC Thin driver, connects to the database, creates a `Statement` object, executes a query, and processes the result set.

The `SELECT` statement retrieves and lists the contents of the `ENAME` column of the `EMP` table.

```
import java.sql.*
import java.math.*
import java.io.*
import java.awt.*

class JdbcTest {
    public static void main (String args []) throws SQLException {
        // Load Oracle driver
        DriverManager.registerDriver (new oracle.jdbc.dnldriver.OracleDriver());

        // Connect to the local database
        Connection conn =
            DriverManager.getConnection ("jdbc:oracle:dnldthin:@myhost:1521:orcl",
                                        "scott", "tiger");

        // Query the employee names
        Statement stmt = conn.createStatement ();
        ResultSet rset = stmt.executeQuery ("SELECT ENAME FROM EMP");

        // Print the name out
        while (rset.next ())
            System.out.println (rset.getString (1));
        // Close the result set, statement, and the connection
        rset.close();
        stmt.close();
        conn.close();
    }
}
```

An Oracle extension to the JDBC drivers is a form of the `getConnection()` method that uses a `Properties` object. The `Properties` object lets you specify user, password, and database information as well as row prefetching and execution batching.

To use the OCI driver in this code, replace the `Connection` statement with:

```
Connection conn = DriverManager.getConnection ("jdbc:oracle:oci8:@MyHostString",
                                             "scott", "tiger");
```

where `MyHostString` is an entry in the `TNSNAMES.ORA` file.

If you are creating an applet, the `getConnection()` and `registerDriver()` strings will be different.

## Java in the RDBMS

The Oracle Database Server stores Java classes as well as PL/SQL subprograms. Except for GUI methods, any Java method can run in the RDBMS as a stored procedure. The following database constructs are supported:

### Functions and Procedures

You write these named blocks and then define them using the `loadjava`, `SQL CREATE FUNCTION`, `CREATE PROCEDURE`, and/or `CREATE PACKAGE` statements. These Java methods can accept arguments and are callable from:

- `SQL CALL` statements.
- Embedded `SQL CALL` statements.
- PL/SQL blocks, subprograms and packages.
- DML statements (`INSERT`, `UPDATE`, `DELETE`, and `SELECT`).
- Oracle development tools such as OCI, Pro\*C/C++ and Oracle Developer.
- Oracle Java interfaces such as JDBC, SQLJ statements, CORBA, and Enterprise Java Beans.

### Database Triggers

A database trigger is a stored procedure that Oracle invokes ("fires") automatically whenever a given DML operation modifies the trigger's table or schema. Triggers allow you to enforce business rules, prevent invalid values from being stored, and take many other actions without the need for you to explicitly call them.

## Why Use Stored Procedures?

- Stored procedures are compiled once, are easy to use and maintain, and require less memory and computing overhead.
- Network bottlenecks are avoided, and response time is improved. Distributed applications are easier to build and use.
- Computation-bound procedures run faster in the server.
- Data access can be controlled by letting users only have stored procedures that execute with their definer's privileges instead of invoker's rights.
- PL/SQL and Java stored procedures can call each other.

- Java in the server follows the Java language specification and can use the SQLJ standard, so that non-Oracle databases are also supported.
- Stored procedures can be reused in different applications as well as different geographic sites.

## JDBC in SQLJ Applications

JDBC code and SQLJ code (see "[Overview of Oracle SQLJ](#)" on page 1-30) interoperates, allowing dynamic SQL statements in JDBC to be used with static SQL statements in SQLJ. A SQLJ iterator class corresponds to the JDBC result set. For more information on JDBC, see *Oracle8i JDBC Developer's Guide and Reference*.

## Overview of Oracle SQLJ

SQLJ is:

- A language specification for embedding static SQL statements in Java source code which has been agreed to by a consortium of database companies, including Oracle, and by Sun, author of Java. The specification has been accepted by ANSI as a software standard.
- A software tool developed by Oracle to the standard, with extensions to the standard to support Oracle8i features. That tool is the subject of this brief overview.

## SQLJ Tool

The Oracle software tool SQLJ has two parts: a translator and a runtime. You execute on any Java VM with a JDBC driver and a SQLJ runtime library.

A SQLJ source file is a Java source file containing embedded static SQL statements. The SQLJ translator is 100% Pure Java and is portable to any standard JDK 1.1 or higher VM.

The Oracle8i SQLJ implementation runs in three steps:

- Translates SQLJ source to Java code with calls to the SQLJ runtime. The SQLJ translator converts the source code to pure Java source code, and can check the syntax and semantics of static SQL statements against a database schema and verify the type compatibility of host variables with SQL types.
- Compiles using the Java compiler.

- Customizes for the target database. SQLJ generates "profile" files with Oracle-specific customizing.

Oracle8i supports SQLJ stored procedures, functions, and triggers which execute in a Java VM integrated with the data server. SQLJ is integrated with Oracle's JDeveloper. Source-level debugging support is available in JDeveloper.

Here is an example of the simplest SQLJ executable statement, which returns one value because `empno` is unique in the `emp` table:

```
String name;
#sql { SELECT ename INTO :name FROM emp WHERE empno=67890 };
System.out.println("Name is " + name + ", employee number = " + empno);
```

Each host variable (or qualified name or complex Java host expression) is preceded by a colon (:). Other SQLJ statements are declarative (declares Java types) and allow you to declare an iterator (a construct related to a database cursor) for queries that retrieve many values:

```
#sql iterator EmpIter (String EmpNam, int EmpNumb);
```

## SQLJ Design Goals

The primary goal is to provide simple extensions to Java to allow rapid development and easy maintenance of Java applications that use embedded SQL to interact with databases.

Specific goals in support of the primary goal are:

- Provide a concise, legible mechanism for database access via static SQL. Most SQL in applications is static. SQLJ provides more concise and less error-prone static SQL constructs than JDBC does.
- Check static SQL at translate time.
- Provide flexible deployment configurations. This makes it possible to implement SQLJ on the client or database side or in the middle tier.
- Support a software standard. SQLJ is an effort of a group of vendors and will be supported by all of them. Applications can access multiple database vendors.
- Provide source code portability. Executables can be used with all of the vendors' DBMSs presuming the code does not rely on any vendor-specific features.

## Strengths of Oracle's SQLJ Implementation

- Uniform programming style for the clients and the servers.
- Integration of the SQLJ translator with JDeveloper, a graphical IDE that provides SQLJ translation, Java compilation, profile customizing, and debugging at the source code level, all in one step.
- SQL Checker module for verification of syntax and semantics at translate-time.
- Oracle type extensions. Datatypes supported are LOBs, ROWIDs, REF CURSORS, VARRAYs, nested tables, user-defined object types, as well as other datatypes such as RAW and NUMBER.

## Comparison of SQLJ with JDBC

JDBC provides a complete dynamic SQL interface from Java to databases. SQLJ fills a complementary role.

JDBC provides fine-grained control of the execution of dynamic SQL from Java, while SQLJ provides a higher level static binding to SQL operations in a specific database schema. Here are some differences:

- SQLJ source code is more concise than equivalent JDBC source code.
- SQLJ uses database connections to type-check static SQL code. JDBC, being a completely dynamic API, does not.
- SQLJ programs allow direct embedding of Java bind expressions within SQL statements. JDBC requires a separate get and/or set call statement for each bind variable and specifies the binding by position number.
- SQLJ provides strong typing of query outputs and return parameters and allows type-checking on calls. JDBC passes values to and from SQL without compile-time type checking.
- SQLJ provides simplified rules for calling SQL stored procedures and functions. The JDBC specification requires a generic call to a stored procedure (or function), *fun*, to have the following syntax (we show SQL92 and Oracle escape syntaxes, which are both allowed):

```
prepStmt.prepareCall("{call fun(?,?)}");           //stored procedure SQL92
prepStmt.prepareCall("{? = call fun(?,?)}");       //stored function SQL92
prepStmt.prepareCall("begin fun(:1,:2);end;");     //stored procedure Oracle
prepStmt.prepareCall("begin :1 := fun(:2,:3);end;");//stored func Oracle
```

SQLJ provides simplified notations:

```

#sql {call fun(param_list) }; //Stored procedure
// Declare x
...
#sql x = {VALUES(fun(param_list)) }; // Stored function
// where VALUES is the SQL construct

```

Here are similarities:

- SQLJ source files can contain JDBC calls. SQLJ and JDBC are interoperable.
- Oracle's JPublisher tool generates custom Java classes to be used in your SQLJ or JDBC application for mappings to Oracle object types and collections.
- Java and PL/SQL stored procedures can be used interchangeably.

## SQLJ Example for Object Types

Here is a simple use of user-defined objects and object refs taken from *Oracle8i SQLJ Developer's Guide and Reference*, where more information on SQLJ is available:

The following items are created using the SQL script below:

- Two object types, PERSON and ADDRESS
- A typed table for PERSON objects
- An EMPLOYEE table that includes an ADDRESS column and two columns of PERSON references

```

SET ECHO ON;
/
/*** Clean up in preparation ***/
DROP TABLE EMPLOYEES
/
DROP TABLE PERSONS
/
DROP TYPE PERSON FORCE
/
DROP TYPE ADDRESS FORCE
/
/*** Create address UDT ***/
CREATE TYPE address AS OBJECT
(
  street      VARCHAR(60),
  city        VARCHAR(30),
  state       CHAR(2),
  zip_code    CHAR(5)

```

```

)
/
/** Create person UDT containing an embedded address UDT ***/
CREATE TYPE person AS OBJECT
(
  name    VARCHAR(30),
  ssn     NUMBER,
  addr    address
)
/
/** Create a typed table for person objects ***/
CREATE TABLE persons OF person
/
/** Create a relational table with two columns that are REFs
    to person objects, as well as a column which is an Address ADT. ***/
CREATE TABLE employees
(
  empnumber          INTEGER PRIMARY KEY,
  person_data       REF person,
  manager            REF person,
  office_addr       address,
  salary             NUMBER
)
/** Insert some data--2 objects into the persons typed table ***/
INSERT INTO persons VALUES (
  person('Wolfgang Amadeus Mozart', 123456,
    address('Am Berg 100', 'Salzburg', 'AT', '10424'))
)
INSERT INTO persons VALUES (
  person('Ludwig van Beethoven', 234567,
    address('Rheinallee', 'Bonn', 'DE', '69234'))
)
/** Put a row in the employees table **/
INSERT INTO employees (empnumber, office_addr, salary) VALUES (
  1001,
  address('500 Oracle Parkway', 'Redwood Shores', 'CA', '94065'),
  50000)
/
/** Set the manager and person REFs for the employee **/
UPDATE employees
  SET manager =
    (SELECT REF(p) FROM persons p WHERE p.name = 'Wolfgang Amadeus Mozart')
/
UPDATE employees
  SET person_data =

```

```

        (SELECT REF(p) FROM persons p WHERE p.name = 'Ludwig van Beethoven')
    /
    COMMIT
    /
    QUIT

```

Next, `JPublisher` is used to generate the `Address` class for mapping to Oracle `ADDRESS` objects. We omit the details.

The following SQLJ sample declares and sets an input host variable of Java type `Address` to update an `ADDRESS` object in a column of the `employees` table. Both before and after the update, the office address is selected into an output host variable of type `Address` and printed for verification.

```

...
// Updating an object

static void updateObject()
{
    Address addr;
    Address new_addr;
    int empno = 1001;

    try {
        #sql {
            SELECT office_addr
            INTO :addr
            FROM employees
            WHERE empnumber = :empno };
        System.out.println("Current office address of employee 1001:");

        printAddressDetails(addr);

        /* Now update the street of address */

        String street = "100 Oracle Parkway";
        addr.setStreet(street);

        /* Put updated object back into the database */

        try {
            #sql {
                UPDATE employees
                SET office_addr = :addr

```

```
        WHERE empnumber = :empno };
System.out.println
    ("Updated employee 1001 to new address at Oracle Parkway.");

/* Select new address to verify update */

try {
    #sql {
        SELECT office_addr
        INTO :new_addr
        FROM employees
        WHERE empnumber = :empno };

    System.out.println("New office address of employee 1001:");
    printAddressDetails(new_addr);

    } catch (SQLException exn) {
    System.out.println("Verification SELECT failed with "+exn); }

    } catch (SQLException exn) {
    System.out.println("UPDATE failed with "+exn); }

    } catch (SQLException exn) {
    System.out.println("SELECT failed with "+exn); }
}
...
```

Note the use of the `setStreet()` accessor method of the `Address` instance. Remember that `JPublisher` provides such accessor methods for all attributes in any custom Java class that it produces.

## SQLJ Stored Procedures in the Server

SQLJ applications can be stored and run in the server. You have the option of translating, compiling, and customizing SQLJ source on a client and loading the generated classes and resources into the server with the `loadjava` utility, typically using a Java archive (`.jar`) file.

Or, you have a second option of loading SQLJ source code into the server, also using `loadjava`, and having it translated and compiled by the server's embedded translator.

## Choosing a Programming Environment

To choose a programming environment for a new development project:

- Review the preceding overviews and the manuals for each environment.
- Read the platform-specific manual that explains which compilers are approved for use with your platforms.
- Remember that PL/SQL stored procedures can be called from code written in any of the languages in this chapter. Java stored procedures are also available from Pro\*C/C++, OCI, and Pro\*COBOL programs. Stored procedures include triggers and object type methods.
- External procedures written in C are also callable from OCI, Java, PL/SQL or SQL. The external procedure itself can call back into the database using either SQL, OCI, or Pro\*C (but not C++).

The following examples illustrate easy choices:

- Pro\*COBOL does not support object types or collection types, while Pro\*C/C++ does.
- SQLJ does not support dynamic SQL the way that JDBC does.

### Use OCI or a Precompiler?

The use of OCI rather than a precompiler is often dictated by technical requirements because some tasks cannot be done solely with precompiler statements:

- OCI provides more detailed control over multiplexing and migrating sessions.
- OCI provides dynamic bind and define using callbacks that can be used for any arbitrary structure, including lists.
- OCI has many calls to handle metadata.
- OCI allows asynchronous event notifications to be received by a client application. It provides a means for clients to generate notifications for propagation to other clients.
- OCI allows DML statements to use arrays to complete as many iterations as possible and then return a batch of errors.
- OCI calls for special purposes include Advanced Queuing, National Language Support, Data Cartridges, and support of the Datetime datatypes.

- OCI calls can be embedded in a Pro\*C/C++ application.

## Built-in Packages and Libraries

Both Java and PL/SQL have built-in packages and libraries. PL/SQL has libraries such as:

### Application Development Packages

- DBMS\_PIPE is used to communicate between sessions.
- DBMS\_ALERT is used to broadcast alerts to users.
- DBMS\_LOCK and DBMS\_TRANSACTION are used for lock and transaction management.
- DBMS\_AQ is used for Advanced Queuing.
- DBMS\_LOB is for your manipulation of large objects.
- DBMS\_ROWID is used for employing ROWIDs.
- UTL\_RAW is for the RAW facility.
- UTL\_REF is for work with REFs.

### Server Management Packages

- DBMS\_SESSION is for session management by DBAs.
- DBMS\_SYSTEM is used to set events for debugging.
- DBMS\_SPACE and DBMS\_SHARED\_POOL obtain space information and reserve shared pool resources.
- DBMS\_JOB is used to schedule jobs in the server.

### Distributed Database Access

These provide access to snapshots, advanced replication, conflict resolution, deferred transactions, and remote procedure calls.

See [Table 9-2, "List of Oracle Supplied Packages"](#).

Java libraries include:

### Core JDK Libraries

These are the libraries such as java.lang, java.io, etc.

### Other Libraries

These are libraries such as Java-based CORBA ORB, EJB (Enterprise Java Beans), Java-based XML Parser, and Java Web Server.

PL/SQL and Java interoperate in the server. You can execute a PL/SQL package from Java or wrap a PL/SQL class with a Java wrapper so that it can be called from distributed CORBA and EJB clients. The following table shows PL/SQL packages and their Java equivalents:

**Table 1–1 PL/SQL and Java Equivalent Software**

PL/SQL Package	Java Equivalent
DBMS_ALERT	Call package with SQLJ or JDBC.
DBMS_DDL	JDBC has this functionality.
DBMS_JOB	Schedule a job that has a Java Stored procedure.
DBMS_LOCK	Call with SQLJ or JDBC.
DBMS_MAIL	Use JavaMail.
DBMS_OUTPUT	Use subclass <code>oracle.aurora.rdbms.OracleDBMSOutputStream</code> or Java stored procedure <code>DBMS_JAVA.SET_STREAMS</code> .
DBMS_PIPE	Call with SQLJ or JDBC.
DBMS_SESSION	Use JDBC to execute an <code>ALTER SESSION</code> statement.
DBMS_SNAPSHOT	Call with SQLJ or JDBC.
DBMS_SQL	Use JDBC.
DBMS_TRANSACTION	Use JDBC to execute an <code>ALTER SESSION</code> statement.
DBMS_UTILITY	Call with SQLJ or JDBC.
UTL_FILE	Grant the <code>JAVAUSERPRIV</code> privilege and then use Java IO entry points.

### Java vs. PL/SQL

Both Java and PL/SQL can be used to build applications in the database and will have future performance improvements. Here are guidelines for their use:

### **PL/SQL is Optimized for Database Access**

PL/SQL uses the same datatypes as SQL. SQL datatypes are thus easier to use and SQL operations are faster than with Java, especially when a large amount of data is involved, when mostly database access is done, or when bulk operations are used.

### **PL/SQL is Integrated with the Database**

PL/SQL is the extension to SQL and uses the same datatypes. PL/SQL has data encapsulation, information hiding, overloading, and exception-handling.

Many optimal capabilities that PL/SQL has had for a long time were not yet implemented for Java in Oracle8i. Examples are autonomous transactions and the dblink facility for remote databases. Code development is usually faster than when using Java.

### **Java is Optimized for Computation**

Java executes much faster than PL/SQL when they are not doing any database access and it is better than PL/SQL at all complex object-oriented and CPU-intensive applications.

Java has inheritance, polymorphism, and component models for developing distributed systems. PL/SQL does not.

### **Java Is Used for Open Distributed Applications**

Java has a richer type system than PL/SQL and is an object-oriented language. Java can use CORBA (which can have many different computer languages in its clients) and EJB. However, PL/SQL packages can also be called from CORBA or EJB clients.

You can run XML tools, the Internet File System, or JavaMail from Java.

# Part II

---

## Designing the Database

This part contains the following chapters:

- Chapter 2, "Managing Schema Objects"
- Chapter 3, "Selecting a Datatype"
- Chapter 4, "Maintaining Data Integrity"
- Chapter 5, "Selecting an Index Strategy"
- Chapter 6, "Speeding Up Index Access with Index-Organized Tables"
- Chapter 7, "Processing SQL Statements"
- Chapter 8, "Dynamic SQL"
- Chapter 9, "Using Procedures and Packages"
- Chapter 10, "External Routines"
- Chapter 11, "Establishing Security Policies"



---

# Managing Schema Objects

This chapter discusses the procedures necessary to create and manage the different types of objects contained in a user's schema. The topics include:

- [Managing Tables](#)
- [Managing Temporary Tables](#)
- [Managing Views](#)
- [Modifying a Join View](#)
- [Managing Sequences](#)
- [Managing Synonyms](#)
- [Creating Multiple Tables and Views in One Operation](#)
- [Naming Schema Objects](#)
- [Renaming Schema Objects](#)
- [Listing Information about Schema Objects](#)

**See Also:**

- Indexes and clusters — [Chapter 5](#)
- Procedures, functions, and packages — [Chapter 9](#)
- Object types — *Oracle8i Application Developer's Guide - Object-Relational Features*
- Dependency information — [Chapter 9](#)
- If you use symmetric replication, then see *Oracle8i Replication* for information on managing schema objects, such as snapshots.

## Managing Tables

A table is the data structure that holds data in a relational database. A table is composed of rows and columns.

A table can represent a single entity that you want to track within your system. This type of a table could represent a list of the employees within your organization, or the orders placed for your company's products.

A table can also represent a relationship between two entities. This type of a table could portray the association between employees and their job skills, or the relationship of products to orders. Within the tables, foreign keys are used to represent relationships.

Although some well designed tables could represent both an entity and describe the relationship between that entity and another entity, most tables should represent either an entity or a relationship. For example, the `EMP_TAB` table describes the employees in a firm, but this table also includes a foreign key column, `DEPTNO`, which represents the relationships of employees to departments.

The following sections explain how to create, alter, and drop tables. Some simple guidelines to follow when managing tables in your database are included.

**See Also:** The *Oracle8i Administrator's Guide* has more suggestions. You should also refer to a text on relational database or table design.

## Designing Tables

Consider the following guidelines when designing your tables:

- Use descriptive names for tables, columns, indexes, and clusters.
- Be consistent in abbreviations and in the use of singular and plural forms of table names and columns.
- Document the meaning of each table and its columns with the `COMMENT` command.
- Normalize each table.
- Select the appropriate datatype for each column.
- Define columns that allow nulls last, to conserve storage space.
- Cluster tables whenever appropriate, to conserve storage space and optimize performance of SQL statements.

Before creating a table, you should also determine whether to use integrity constraints. Integrity constraints can be defined on the columns of a table to enforce the business rules of your database automatically.

**See Also:** See [Chapter 4, "Maintaining Data Integrity"](#) for guidelines.

## Creating Tables

To create a table, use the SQL command `CREATE TABLE`. For example, if the user `SCOTT` issues the following statement, he creates a non-clustered table named `Emp_tab` in his schema that is physically stored in the `USERS` tablespace. Notice that integrity constraints are defined on several columns of the table.

```
CREATE TABLE Emp_tab (  
    Empno      NUMBER(5) PRIMARY KEY,  
    Ename      VARCHAR2(15) NOT NULL,  
    Job        VARCHAR2(10),  
    Mgr        NUMBER(5),  
    Hiredate   DATE DEFAULT (sysdate),  
    Sal        NUMBER(7,2),  
    Comm       NUMBER(7,2),  
    Deptno     NUMBER(3) NOT NULL,  
              CONSTRAINT dept_afkey REFERENCES Dept_tab(Deptno))  
  
PCTFREE 10  
PCTUSED 40  
TABLESPACE users  
STORAGE ( INITIAL 50K  
          NEXT 50K  
          MAXEXTENTS 10  
          PCTINCREASE 25 );
```

### Managing the Space Usage of Data Blocks

The following sections explain the `PCTFREE` and `PCTUSED` parameters help you to:

- Increase the performance of writing and retrieving a data or index segment
- Decrease the amount of unused space in data blocks
- Decrease the amount of row chaining between data blocks

### Specifying `PCTFREE`

The `PCTFREE` default is 10 percent; any integer from 0 to 99 is acceptable, as long as the sum of `PCTFREE` and `PCTUSED` does not exceed 100. (If `PCTFREE` is set to 99, then Oracle puts at least one row in each block, regardless of row size. If the rows are very small and blocks very large, then even more than one row might fit.)

A lower `PCTFREE`:

- Reserves less room for updates to existing table rows
- Allows inserts to fill the block more completely

- Might save space, because the total data for a table or index is stored in fewer blocks (more rows or entries per block)
- Increases processing costs because blocks frequently need to be reorganized as their free space area becomes filled with new or updated data
- Potentially increases processing costs and space required if updates to rows or index entries cause rows to grow and span blocks (because `UPDATE`, `DELETE`, and `SELECT` statements might need to read more blocks for a given row and because chained row pieces contain references to other pieces)

A higher `PCTFREE`:

- Reserves more room for future updates to existing table rows
- Might require more blocks for the same amount of inserted data (inserting fewer rows per block)
- Lessens processing costs, because blocks infrequently need reorganization of their free space area
- Might improve update performance, because Oracle must chain row pieces less frequently, if ever

In setting `PCTFREE`, you should understand the nature of the table or index data. Updates can cause rows to grow. When using `NUMBER`, `VARCHAR2`, `LONG`, or `LONG RAW`, new values might not be the same size as values they replace. If there are many updates in which data values get longer, then increase `PCTFREE`; if updates to rows do not affect the total row width, then `PCTFREE` can be low.

Your goal is to find a satisfactory trade-off between densely packed data (low `PCTFREE`, full blocks) and good update performance (high `PCTFREE`, less-full blocks).

`PCTFREE` also affects the performance of a given user's queries on tables with uncommitted transactions belonging to other users. Assuring read consistency might cause frequent reorganization of data in blocks that have little free space.

**PCTFREE for Non-Clustered Tables** If the data in the rows of a non-clustered table is likely to increase in size over time, then reserve space for these updates. If you do not reserve room for updates, then updated rows are likely to be chained between blocks, reducing I/O performance associated with these rows.

**PCTFREE for Clustered Tables** The discussion for non-clustered tables also applies to clustered tables. However, if `PCTFREE` is reached, then new rows from *any* table

contained in the same cluster key go into a new data block chained to the existing cluster key.

**PCTFREE for Indexes** Indexes infrequently require the use of free space for updates to index data. Therefore, the `PCTFREE` value for index segment data blocks is normally very low (for example, 5 or less).

### Specifying `PCTUSED`

Once the percentage of free space in a data block reaches `PCTFREE`, no new rows are inserted in that block until the percentage of space used falls below `PCTUSED`. Oracle tries to keep a data block at least `PCTUSED` full. The percent is of block space available for data after overhead is subtracted from total space.

The default for `PCTUSED` is 40 percent; any integer between 0 and 99, inclusive, is acceptable as long as the sum of `PCTUSED` and `PCTFREE` does not exceed 100.

A lower `PCTUSED`:

- Usually keeps blocks less full than a higher `PCTUSED`
- Reduces processing costs incurred during `UPDATE` and `DELETE` statements for moving a block to the free list when the block has fallen below that percentage of usage
- Increases the unused space in a database

A higher `PCTUSED`:

- Usually keeps blocks fuller than a lower `PCTUSED`
- Improves space efficiency
- Increases processing cost during `INSERTs` and `UPDATEs`

### Choosing Associated `PCTUSED` and `PCTFREE` Values

If you decide not to use the default values for `PCTFREE` and `PCTUSED`, then use the following guidelines.

- The sum of `PCTFREE` and `PCTUSED` must be equal to or less than 100.
- If the sum is less than 100, then the ideal compromise of space utilization and I/O performance is a sum of `PCTFREE` and `PCTUSED` that differs from 100 by the percentage of space in the available block that an average row occupies. For example, assume that the data block size is 2048 bytes, minus 100 bytes of overhead, leaving 1948 bytes available for data. If an average row requires 195

---

bytes, or 10% of 1948, then an appropriate combination of `PCTUSED` and `PCTFREE` that sums to 90% would make the best use of database space.

- If the sum equals 100, then Oracle attempts to keep no more than `PCTFREE` free space, and the processing costs are highest.
- Fixed block overhead is not included in the computation of `PCTUSED` or `PCTFREE`.
- The smaller the difference between 100 and the sum of `PCTFREE` and `PCTUSED` (as in `PCTUSED` of 75, `PCTFREE` of 20), the more efficient space usage is at some performance cost.

### Examples of Choosing `PCTFREE` and `PCTUSED` Values

The following examples illustrate correctly specifying values for `PCTFREE` and `PCTUSED` in given scenarios.

#### Example1

---

Scenario:	Common activity includes <code>UPDATE</code> statements that increase the size of the rows. Performance is important.
Settings:	<code>PCTFREE = 20</code> <code>PCTUSED = 40</code>
Explanation:	<code>PCTFREE</code> is set to 20 to allow enough room for rows that increase in size as a result of updates. <code>PCTUSED</code> is set to 40 so that less processing is done during high update activity, thus improving performance.

---

### Example2

---

Scenario:	Most activity includes INSERT and DELETE statements, and UPDATE statements that do not increase the size of affected rows. Performance is important.
Settings:	PCTFREE = 5 PCTUSED = 60
Explanation:	PCTFREE is set to 5 because most UPDATE statements do not increase row sizes. PCTUSED is set to 60 so that space freed by DELETE statements is used relatively soon, yet the amount of processing is minimized.

---

### Example3

---

Scenario:	The table is very large; therefore, storage is a primary concern. Most activity includes read-only transactions; therefore, query performance is important.
Settings:	PCTFREE = 5 PCTUSED = 90
Explanation:	PCTFREE is set to 5, because UPDATE statements are rarely issued. PCTUSED is set to 90, so that more space per block is used to store table data. This setting for PCTUSED reduces the number of data blocks required to store the table's data and decreases the average number of data blocks to scan for queries, thereby increasing the performance of queries.

---

### Privileges Required to Create a Table

To create a new table in your schema, you must have the CREATE TABLE system privilege. To create a table in another user's schema, you must have the CREATE ANY TABLE system privilege. Additionally, the owner of the table must have a quota for the tablespace that contains the table, or the UNLIMITED TABLESPACE system privilege.

## Altering Tables

Alter a table in an Oracle database for any of the following reasons:

- To add one or more new columns to the table
- To add one or more integrity constraints to a table
- To modify an existing column's definition (datatype, length, default value, and NOT NULL integrity constraint)
- To modify data block space usage parameters (PCTFREE, PCTUSED)
- To modify transaction entry settings (INITRANS, MAXTRANS)
- To modify storage parameters (NEXT, PCTINCREASE, etc.)
- To enable or disable integrity constraints associated with the table
- To drop integrity constraints associated with the table

When altering the column definitions of a table, you can only increase the length of an existing column, unless the table has no records. You can also decrease the length of a column in an empty table. For columns of datatype CHAR, increasing the length of a column might be a time consuming operation that requires substantial additional storage, especially if the table contains many rows. This is because the CHAR value in each row must be blank-padded to satisfy the new column length.

If you change the datatype (for example, from VARCHAR2 to CHAR), then the data in the column does not change. However, the length of new CHAR columns might change, due to blank-padding requirements.

Use the SQL command ALTER TABLE to alter a table. For example:

```
ALTER TABLE Emp_tab
  PCTFREE 30
  PCTUSED 60;
```

Altering a table has the following implications:

- If a new column is added to a table, then the column is initially null. You can add a column with a NOT NULL constraint to a table only if the table does not contain any rows.
- If a view or PL/SQL program unit depends on a base table, then the alteration of the base table might affect the dependent object, and always invalidates the dependent object.

### Privileges Required to Alter a Table

To alter a table, the table must be contained in your schema, or you must have either the `ALTER` object privilege for the table or the `ALTER ANY TABLE` system privilege.

## Dropping Tables

Use the SQL command `DROP TABLE` to drop a table. For example, the following statement drops the `EMP_TAB` table:

```
DROP TABLE Emp_tab;
```

If the table that you are dropping contains any primary or unique keys referenced by foreign keys of other tables, and if you intend to drop the `FOREIGN KEY` constraints of the child tables, then include the `CASCADE` option in the `DROP TABLE` command. For example:

```
DROP TABLE Emp_tab CASCADE CONSTRAINTS;
```

Dropping a table has the following effects:

- The table definition is removed from the data dictionary. All rows of the table are then inaccessible.
- All indexes and triggers associated with the table are dropped.
- All views and PL/SQL program units that depend on a dropped table remain, but become invalid (not usable).
- All synonyms for a dropped table remain, but return an error when used.
- All extents allocated for a non-clustered table that is dropped are returned to the free space of the tablespace and can be used by any other object requiring new extents.
- All rows corresponding to a clustered table are deleted from the blocks of the cluster.
- If the table is a master table for snapshots, then Oracle does not drop the snapshots, but does drop the snapshot log. The snapshots can still be used, but they cannot be refreshed unless the table is re-created.

If you want to delete all of the rows of a table, but keep the table definition, then you should use the `TRUNCATE TABLE` command.

**See Also:** *Oracle8i Administrator's Guide.*

### Privileges Required to Drop a Table

To drop a table, the table must be contained in your schema or you must have the `DROP ANY TABLE` system privilege.

## Managing Temporary Tables

A temporary table has a definition or structure that persists like that of a regular table, but the data it contains exists only for the duration of a transaction or session. Oracle8i allows you to create temporary tables to hold session-private data. You specify whether the data is specific to a session or to a transaction.

Here are a few examples of when temporary tables can be useful:

- A Web-based airlines reservations application allows you, as a customer, to create several optional itineraries. As you develop each itinerary, the application places the data in a row of a single temporary table. As you modify each itinerary, the application updates that row accordingly. When you ultimately decide which itinerary you want to use, the application moves the row for that itinerary to a persistent table.  
  
During your session, the data you enter is private. When you end your session, the optional itineraries you developed are dropped.
- Several sales agents for a large bookseller use a single temporary table concurrently while taking customer orders over the phone. To enter and modify customer orders, each agent accesses the table in a session that is unavailable to the other agents. When the agent closes a session, the data from that session is automatically dropped, but the table structure persists for the other agents to use.
- An administrator uses temporary tables to improve performance when running an otherwise complex and expensive query. To do this, the administrator caches the values from a more complex query in temporary tables, then runs SQL statements, such as joins, against those temporary tables. For a thorough explanation of how this can be done, see ["Example 2: Using Temporary Tables to Improve Performance"](#) on page 2-13.

## Creating Temporary Tables

You create a temporary table by using special ANSI keywords. You specify the data as *session*-specific by using the `ON COMMIT PRESERVE ROWS` keywords. You specify the data as *transaction*-specific by using the `ON COMMIT DELETE ROWS` keywords.

### **Example 2-1** *Creating a Session-Specific Temporary Table*

```
CREATE GLOBAL TEMPORARY TABLE ...  
    [ON COMMIT PRESERVE ROWS ]
```

### **Example 2-2** *Creating a Transaction-Specific Temporary Table*

```
CREATE GLOBAL TEMPORARY TABLE ...  
    [ON COMMIT DELETE ROWS ]
```

## Using Temporary Tables

You can create indexes on temporary tables as you would on permanent tables.

For a *session*-specific temporary table, a session gets bound to the temporary table with the first insert in the table in the session. This binding goes away at the end of the session or by issuing a `TRUNCATE` of the table in the session.

For a *transaction*-specific temporary table, a session gets bound to the temporary table with the first insert in the table in the transaction. The binding goes away at the end of the transaction.

DDL operations (except `TRUNCATE`) are allowed on an existing temporary table only if no session is currently bound to that temporary table.

Unlike permanent tables, temporary tables and their indexes do not automatically allocate a segment when they are created. Instead, segments are allocated when the first `INSERT` (or `CREATE TABLE AS SELECT`) is performed. This means that if a `SELECT`, `UPDATE`, or `DELETE` is performed before the first `INSERT`, the table appears to be empty.

Temporary segments are deallocated at the end of the transaction for transaction-specific temporary tables and at the end of the session for session-specific temporary tables.

If you rollback a transaction, the data you entered is lost, although the table definition persists.

You cannot create a table that is simultaneously both transaction- and session-specific.

A transaction-specific temporary table allows only one transaction at a time. If there are several autonomous transactions in a single transaction scope, each autonomous transaction can use the table only as soon as the previous one commits.

Because the data in a temporary table is, by definition, temporary, backup and recovery of a temporary table's data is not available in the event of a system failure. To prepare for such a failure, you should develop alternative methods for preserving temporary table data.

## Examples: Using Temporary Tables

### Example 1: A Session-specific Temporary Table

The following statement creates a session-specific temporary table, `FLIGHT_SCHEDULE`, for use in an automated airline reservation scheduling system. Each client has its own session and can store temporary schedules. The temporary schedules are deleted at the end of the session.

```
CREATE GLOBAL TEMPORARY TABLE flight_schedule (
    startdate DATE,
    enddate DATE,
    cost NUMBER)
ON COMMIT PRESERVE ROWS;
```

### Example 2: Using Temporary Tables to Improve Performance

You can use temporary tables to improve performance when you run complex queries. Running multiple such queries is relatively slow because the tables are accessed multiple times for each returned row. It is faster to cache the values from a complex query in a temporary table, then run the queries against the temporary table.

For example, even with a view like this defined to simplify further queries, the queries against the view may be slow because the contents of the view are recalculated each time:

```
CREATE OR REPLACE VIEW Profile_values_view AS
SELECT d.Profile_option_name, d.Profile_option_id, Profile_option_value,
       u.User_name, Level_id, Level_code
FROM Profile_definitions d, Profile_values v, Profile_users u
WHERE d.Profile_option_id = v.Profile_option_id
      AND ((Level_code = 'USER' AND Level_id = U.User_id) OR
           (Level_code = 'DEPARTMENT' AND Level_id = U.Department_id) OR
           (Level_code = 'SITE'))
```

```
AND NOT EXISTS (SELECT 1 FROM PROFILE_VALUES P
                WHERE P.PROFILE_OPTION_ID = V.PROFILE_OPTION_ID
                AND ((Level_code = 'USER' AND
                    level_id = u.User_id) OR
                    (Level_code = 'DEPARTMENT' AND
                    level_id = u.Department_id) OR
                    (Level_code = 'SITE'))
                AND INSTR('USERDEPARTMENTSITE', v.Level_code) >
                INSTR('USERDEPARTMENTSITE', p.Level_code));
```

A temporary table allows us to run the computation once, and cache the result in later SQL queries and joins:

```
CREATE GLOBAL TEMPORARY TABLE Profile_values_temp
(
    Profile_option_name  VARCHAR(60)  NOT NULL,
    Profile_option_id    NUMBER(4)    NOT NULL,
    Profile_option_value VARCHAR2(20) NOT NULL,
    Level_code           VARCHAR2(10) ,
    Level_id             NUMBER(4)    ,
    CONSTRAINT Profile_values_temp_pk
        PRIMARY KEY (Profile_option_id)
) ON COMMIT PRESERVE ROWS ORGANIZATION INDEX;

INSERT INTO Profile_values_temp
(Profile_option_name, Profile_option_id, Profile_option_value,
 Level_code, Level_id)
SELECT Profile_option_name, Profile_option_id, Profile_option_value,
        Level_code, Level_id
FROM Profile_values_view;
COMMIT;
```

Now the temporary table can be used to speed up queries, and the results cached in the temporary table are freed automatically by the database when the session ends.

## Managing Views

A *view* is a logical representation of another table or combination of tables. A view derives its data from the tables on which it is based. These tables are called *base tables*. Base tables might in turn be actual tables or might be views themselves.

All operations performed on a view actually affect the base table of the view. You can use views in almost the same way as tables. You can query, update, insert into, and delete from views, just as you can standard tables.

Views can provide a different representation (such as subsets or supersets) of the data that resides within other tables and views. Views are very powerful because they allow you to tailor the presentation of data to different types of users.

The following sections explain how to create, replace, and drop views using SQL commands.

## Creating Views

Use the SQL command `CREATE VIEW` to create a view. For example, the following statement creates a view on a subset of data in the `EMP_TAB` table:

```
CREATE VIEW Sales_staff AS
  SELECT Empno, Ename, Deptno
  FROM Emp_tab
  WHERE Deptno = 10
  WITH CHECK OPTION CONSTRAINT Sales_staff_cnst;
```

The object names are resolved when the view is created or when the program containing the SQL is compiled, relative to the schema of the view owner.

You can define views with any query that references tables, snapshots, or other views; however, the query that defines a view cannot contain the `ORDER BY` or `FOR UPDATE` clauses.

The query that defines the `SALES_STAFF` view references only rows in department 10. Furthermore, `WITH CHECK OPTION` creates the view with the constraint that `INSERT` and `UPDATE` statements issued against the view are not allowed to create or result in rows that the view cannot select.

Considering the example above, the following `INSERT` statement successfully inserts a row into the `EMP_TAB` table via the `SALES_STAFF` view:

```
INSERT INTO Sales_staff VALUES (7584, 'OSTER', 10);
```

However, the following `INSERT` statement is rolled back and returns an error because it attempts to insert a row for department number 30, which could not be selected using the `SALES_STAFF` view:

```
INSERT INTO Sales_staff VALUES (7591, 'WILLIAMS', 30);
```

The following statement creates a view that joins data from the `Emp_tab` and `Dept_tab` tables:

```
CREATE VIEW Division1_staff AS
  SELECT Ename, Empno, Job, Dname
  FROM Emp_tab, Dept_tab
  WHERE Emp_tab.Deptno IN (10, 30)
  AND Emp_tab.Deptno = Dept_tab.Deptno;
```

The `Division1_staff` view is defined by a query that joins information from the `Emp_tab` and `Dept_tab` tables. The `WITH CHECK OPTION` is not specified in the `CREATE VIEW` statement because rows cannot be inserted into or updated in a view defined with a query that contains a join that uses the `WITH CHECK OPTION`.

### Expansion of Defining Queries at View Creation Time

In accordance with the ANSI/ISO standard, Oracle expands any wildcard in a top-level view query into a column list when a view is created and stores the resulting query in the data dictionary; any subqueries are left intact. The column names in an expanded column list are enclosed in quote marks to account for the possibility that the columns of the base object were originally entered with quotes and require them for the query to be syntactically correct.

As an example, assume that the `Dept_view` view is created as follows:

```
CREATE VIEW Dept_view AS SELECT * FROM scott.Dept_tab;
```

Oracle stores the defining query of the `Dept_view` view as

```
SELECT "DEPTNO", "DNAME", "LOC" FROM scott.Dept_tab;
```

Views created with errors do not have wildcards expanded. However, if the view is eventually compiled without errors, then wildcards in the defining query are expanded.

### Creating Views with Errors

A view can be created even if the defining query of the view cannot be executed, as long as the `CREATE VIEW` command has no syntax errors. We call such a view a *view*

*with errors.* For example, if a view refers to a non-existent table or an invalid column of an existing table, or if the owner of the view does not have the required privileges, then the view can still be created and entered into the data dictionary.

You can only create a view with errors by using the `FORCE` option of the `CREATE VIEW` command:

```
CREATE FORCE VIEW AS ...;
```

When a view is created with errors, Oracle returns a message and leaves the status of the view as `INVALID`. If conditions later change so that the query of an invalid view can be executed, then the view can be recompiled and become valid. Oracle dynamically compiles the invalid view if you attempt to use it.

### Privileges Required to Create a View

To create a view, you must have been granted the following privileges:

- You must have the `CREATE VIEW` system privilege to create a view in your schema, or the `CREATE ANY VIEW` system privilege to create a view in another user's schema. These privileges can be acquired explicitly or via a role.
- The **owner** of the view must be explicitly granted the necessary privileges to access all objects referenced within the definition of the view; the owner cannot obtain the required privileges through roles. Also, the functionality of the view is dependent on the privileges of the view's owner. For example, if you (the view owner) are granted only the `INSERT` privilege for Scott's `EMP_TAB` table, then you can create a view on his `EMP_TAB` table, but you can only use this view to insert new rows into the `EMP_TAB` table.
- If the view owner intends to grant access to the view to other users, then the owner must receive the object privileges to the base objects with the `GRANT OPTION` or the system privileges with the `ADMIN OPTION`; if not, then the view owner has insufficient privileges to grant access to the view to other users.

## Replacing Views

To alter the definition of a view, you must replace the view using one of the following methods:

- A view can be dropped and then re-created. When a view is dropped, all grants of corresponding view privileges are revoked from roles and users. After the view is re-created, necessary privileges must be regranted.

- A view can be replaced by redefining it with a `CREATE VIEW` statement that contains the `OR REPLACE` option. This option replaces the current definition of a view, but preserves the present security authorizations.

For example, assume that you create the `SALES_STAFF` view, as given in a previous example. You also grant several object privileges to roles and other users. However, now you realize that you must redefine the `SALES_STAFF` view to correct the department number specified in the `WHERE` clause of the defining query, because it should have been 30. To preserve the grants of object privileges that you have made, you can replace the current version of the `SALES_STAFF` view with the following statement:

```
CREATE OR REPLACE VIEW Sales_staff AS
    SELECT Empno, Ename, Deptno
    FROM Emp_tab
    WHERE Deptno = 30
    WITH CHECK OPTION CONSTRAINT Sales_staff_cnst;
```

Replacing a view has the following effects:

- Replacing a view replaces the view's definition in the data dictionary. All underlying objects referenced by the view are not affected.
- If previously defined but not included in the new view definition, then the constraint associated with the `WITH CHECK OPTION` for a view's definition is dropped.
- All views and PL/SQL program units dependent on a replaced view become invalid.

### Privileges Required to Replace a View

To replace a view, you must have all of the privileges necessary to drop the view, as well as all of those required to create the view.

## Using Views

Views can be queried in the same manner as tables. For example, to query the `Division1_staff` view, enter a valid `SELECT` statement that references the view:

```
SELECT * FROM Division1_staff;
```

ENAME	EMPNO	JOB	DNAME
CLARK	7782	MANAGER	ACCOUNTING
KING	7839	PRESIDENT	ACCOUNTING

MILLER	7934	CLERK	ACCOUNTING
ALLEN	7499	SALESMAN	SALES
WARD	7521	SALESMAN	SALES
JAMES	7900	CLERK	SALES
TURNER	7844	SALESMAN	SALES
MARTIN	7654	SALESMAN	SALES
BLAKE	7698	MANAGER	SALES

With some restrictions, rows can be inserted into, updated in, or deleted from a base table using a view. The following statement inserts a new row into the EMP\_TAB table using the SALES\_STAFF view:

```
INSERT INTO Sales_staff
VALUES (7954, 'OSTER', 30);
```

Restrictions on DML operations for views use the following criteria in the order listed:

1. If a view is defined by a query that contains SET or DISTINCT operators, a GROUP BY clause, or a group function, then rows cannot be inserted into, updated in, or deleted from the base tables using the view.
2. If a view is defined with WITH CHECK OPTION, then a row cannot be inserted into, or updated in, the base table (using the view), if the view cannot select the row from the base table.
3. If a NOT NULL column that does not have a DEFAULT clause is omitted from the view, then a row cannot be inserted into the base table using the view.
4. If the view was created by using an expression, such as DECODE(deptno, 10, "SALES", ...), then rows cannot be inserted into or updated in the base table using the view.

The constraint created by WITH CHECK OPTION of the SALES\_STAFF view only allows rows that have a department number of 10 to be inserted into, or updated in, the EMP\_TAB table. Alternatively, assume that the SALES\_STAFF view is defined by the following statement (that is, excluding the DEPTNO column):

```
CREATE VIEW Sales_staff AS
SELECT Empno, Ename
FROM Emp_tab
WHERE Deptno = 10
WITH CHECK OPTION CONSTRAINT Sales_staff_cnst;
```

Considering this view definition, you can update the EMPNO or ENAME fields of existing records, but you cannot insert rows into the EMP\_TAB table via the SALES\_

STAFF view because the view does not let you alter the DEPTNO field. If you had defined a DEFAULT value of 10 on the DEPTNO field, then you could perform inserts.

**Referencing Invalid Views** When a user attempts to reference an invalid view, Oracle returns an error message to the user:

```
ORA-04063: view 'view_name' has errors
```

This error message is returned when a view exists but is unusable due to errors in its query (whether it had errors when originally created or it was created successfully but became unusable later because underlying objects were altered or dropped).

### Privileges Required to Use a View

To issue a query or an INSERT, UPDATE, or DELETE statement against a view, you must have the SELECT, INSERT, UPDATE, or DELETE object privilege for the view, respectively, either explicitly or via a role.

## Dropping Views

Use the SQL command DROP VIEW to drop a view. For example:

```
DROP VIEW Sales_staff;
```

### Privileges Required to Drop a View

You can drop any view contained in your schema. To drop a view in another user's schema, you must have the DROP ANY VIEW system privilege.

## Modifying a Join View

The Oracle Server allows you, with some restrictions, to modify views that involve joins. Consider the following simple view:

```
CREATE VIEW Emp_view AS
  SELECT Ename, Empno, deptno FROM Emp_tab;
```

This view does not involve a join operation. If you issue the SQL statement:

```
UPDATE Emp_view SET Ename = 'CAESAR' WHERE Empno = 7839;
```

then the EMP\_TAB base table that underlies the view changes, and employee 7839's name changes from KING to CAESAR in the EMP\_TAB table.

However, if you create a view that involves a join operation, such as:

```
CREATE VIEW Emp_dept_view AS
  SELECT e.Empno, e.Ename, e.Deptno, e.Sal, d.Dname, d.Loc
  FROM Emp_tab e, Dept_tab d /* JOIN operation */
  WHERE e.Deptno = d.Deptno
  AND d.Loc IN ('DALLAS', 'NEW YORK', 'BOSTON');
```

then there are restrictions on modifying either the EMP\_TAB or the DEPT\_TAB base table through this view, for example, using a statement such as:

```
UPDATE Emp_dept_view SET Ename = 'JOHNSON'
  WHERE Ename = 'SMITH';
```

A *modifiable join view* is a view that contains more than one table in the top-level FROM clause of the SELECT statement, and that does *not* contain any of the following:

- DISTINCT operator
- Aggregate functions: AVG, COUNT, GLB, MAX, MIN, STDDEV, SUM, or VARIANCE
- Set operations: UNION, UNION ALL, INTERSECT, MINUS
- GROUP BY or HAVING clauses
- START WITH or CONNECT BY clauses
- ROWNUM pseudocolumn

A further restriction on which join views are modifiable is that if a view is a join on other nested views, then the other nested views must be mergeable into the top level view.

**See Also:** See *Oracle8i Concepts* for more information about mergeable views.

### Example Tables

The examples in this section use the EMP\_TAB and DEPT\_TAB tables. However, the examples work only if you explicitly define the primary and foreign keys in these tables, or define unique indexes. Here are the appropriately constrained table definitions for EMP\_TAB and DEPT\_TAB:

```
CREATE TABLE Dept_tab (
    Deptno    NUMBER(4) PRIMARY KEY,
    Dname     VARCHAR2(14),
    Loc       VARCHAR2(13));

CREATE TABLE Emp_tab (
    Empno     NUMBER(4) PRIMARY KEY,
    Ename     VARCHAR2(10),
    Job       varchar2(9),
    Mgr       NUMBER(4),
    Hiredate  DATE,
    Sal       NUMBER(7,2),
    Comm      NUMBER(7,2),
    Deptno    NUMBER(2),
    FOREIGN KEY (Deptno) REFERENCES Dept_tab(Deptno));
```

You could also omit the primary and foreign key constraints listed above, and create a UNIQUE INDEX on DEPT\_TAB (DEPTNO) to make the following examples work.

### Key-Preserved Tables

The concept of a *key-preserved table* is fundamental to understanding the restrictions on modifying join views. A table is key preserved if every key of the table can also be a key of the result of the join. So, a key-preserved table has its keys preserved through a join.

**Note:**

- It is not necessary that the key or keys of a table be selected for it to be key preserved. It is sufficient that if the key or keys were selected, then they would also be key(s) of the result of the join.
- The key-preserving property of a table does not depend on the actual data in the table. It is, rather, a property of its schema and not of the data in the table. For example, if in the EMP\_TAB table there was at most one employee in each department, then DEPT\_TAB.DEPTNO would be unique in the result of a join of EMP\_TAB and DEPT\_TAB, but DEPT\_TAB would still not be a key-preserved table.

If you SELECT all rows from EMP\_DEPT\_VIEW defined in the ["Modifying a Join View"](#) section, then the results are:

EMPNO	ENAME	DEPTNO	DNAME	LOC
7782	CLARK	10	ACCOUNTING	NEW YORK
7839	KING	10	ACCOUNTING	NEW YORK
7934	MILLER	10	ACCOUNTING	NEW YORK
7369	SMITH	20	RESEARCH	DALLAS
7876	ADAMS	20	RESEARCH	DALLAS
7902	FORD	20	RESEARCH	DALLAS
7788	SCOTT	20	RESEARCH	DALLAS
7566	JONES	20	RESEARCH	DALLAS

8 rows selected.

In this view, EMP\_TAB is a key-preserved table, because EMPNO is a key of the EMP\_TAB table, and also a key of the result of the join. DEPT\_TAB is *not* a key-preserved table, because although DEPTNO is a key of the DEPT\_TAB table, it is not a key of the join.

## Rule for DML Statements on Join Views

Any UPDATE, INSERT, or DELETE statement on a join view can modify *only one underlying base table*.

## UPDATE Statements

The following example shows an UPDATE statement that successfully modifies the EMP\_DEPT\_VIEW view:

```
UPDATE Emp_dept_view
  SET Sal = Sal * 1.10
  WHERE Deptno = 10;
```

The following UPDATE statement would be disallowed on the EMP\_DEPT\_VIEW view:

```
UPDATE Emp_dept_view
  SET Loc = 'BOSTON'
  WHERE Ename = 'SMITH';
```

This statement fails with an ORA-01779 error ("cannot modify a column which maps to a non key-preserved table"), because it attempts to modify the underlying DEPT\_TAB table, and the DEPT\_TAB table is not key preserved in the EMP\_DEPT view.

In general, all modifiable columns of a join view must map to columns of a key-preserved table. If the view is defined using the WITH CHECK OPTION clause, then all join columns and all columns of repeated tables are not modifiable.

So, for example, if the EMP\_DEPT view were defined using WITH CHECK OPTION, then the following UPDATE statement would fail:

```
UPDATE Emp_dept_view
  SET Deptno = 10
  WHERE Ename = 'SMITH';
```

The statement fails because it is trying to update a join column.

## DELETE Statements

You can delete from a join view provided there is *one and only one* key-preserved table in the join.

The following DELETE statement works on the EMP\_DEPT view:

```
DELETE FROM Emp_dept_view
  WHERE Ename = 'SMITH';
```

This DELETE statement on the EMP\_DEPT view is legal because it can be translated to a DELETE operation on the base EMP\_TAB table, and because the EMP\_TAB table is the only key-preserved table in the join.

In the following view, a DELETE operation cannot be performed on the view because both E1 and E2 are key-preserved tables:

```
CREATE VIEW emp_emp AS
  SELECT e1.Ename, e2.Empno, e1.Deptno
  FROM Emp_tab e1, Emp_tab e2
  WHERE e1.Empno = e2.Empno;
  WHERE e1.Empno = e2.Empno;
```

If a view is defined using the WITH CHECK OPTION clause and the key-preserved table is repeated, then rows cannot be deleted from such a view. For example:

```
CREATE VIEW Emp_mgr AS
  SELECT e1.Ename, e2.Ename Mname
  FROM Emp_tab e1, Emp_tab e2
  WHERE e1.mgr = e2.Empno
  WITH CHECK OPTION;
```

No deletion can be performed on this view because the view involves a self-join of the table that is key preserved.

## INSERT Statements

The following INSERT statement on the EMP\_DEPT view succeeds, because only one key-preserved base table is being modified (EMP\_TAB), and 40 is a valid DEPTNO in the DEPT\_TAB table (thus satisfying the FOREIGN KEY integrity constraint on the EMP\_TAB table).

```
INSERT INTO Emp_dept (Ename, Empno, Deptno)
  VALUES ('KURODA', 9010, 40);
```

The following INSERT statement fails for the same reason: This UPDATE on the base EMP\_TAB table would fail: the FOREIGN KEY integrity constraint on the EMP\_TAB table is violated.

```
INSERT INTO Emp_dept (Ename, Empno, Deptno)
  VALUES ('KURODA', 9010, 77);
```

The following INSERT statement fails with an ORA-01776 error ("cannot modify more than one base table through a view").

```
INSERT INTO Emp_dept (Ename, Empno, Deptno)
  VALUES (9010, 'KURODA', 'BOSTON');
```

An INSERT cannot, implicitly or explicitly, refer to columns of a non-key-preserved table. If the join view is defined using the WITH CHECK OPTION clause, then you cannot perform an INSERT to it.

## Using the UPDATABLE\_COLUMNS Views

Three views you can use for modifying join views are shown in [Table 2-1](#).

**Table 2-1** UPDATABLE\_COLUMNS Views

View Name	Description
USER_UPDATABLE_COLUMNS	Shows all columns in all tables and views in the user's schema that are modifiable
DBA_UPDATABLE_COLUMNS	Shows all columns in all tables and views in the DBA schema that are modifiable
ALL_UPDATABLE_VIEWS	Shows all columns in all tables and views that are modifiable

## Outer Joins

Views that involve outer joins are modifiable in some cases. For example:

```
CREATE VIEW Emp_dept_oj1 AS
  SELECT Empno, Ename, e.Deptno, Dname, Loc
  FROM Emp_tab e, Dept_tab d
  WHERE e.Deptno = d.Deptno (+);
```

The statement:

```
SELECT * FROM Emp_dept_oj1;
```

Results in:

EMPNO	ENAME	DEPTNO	DNAME	LOC
7369	SMITH	40	OPERATIONS	BOSTON
7499	ALLEN	30	SALES	CHICAGO
7566	JONES	20	RESEARCH	DALLAS
7654	MARTIN	30	SALES	CHICAGO
7698	BLAKE	30	SALES	CHICAGO
7782	CLARK	10	ACCOUNTING	NEW YORK
7788	SCOTT	20	RESEARCH	DALLAS
7839	KING	10	ACCOUNTING	NEW YORK
7844	TURNER	30	SALES	CHICAGO

```

7876  ADAMS      20    RESEARCH    DALLAS
7900  JAMES      30    SALES      CHICAGO
7902  FORD       20    RESEARCH    DALLAS
7934  MILLER     10    ACCOUNTING  NEW YORK
7521  WARD       30    SALES      CHICAGO
14 rows selected.

```

Columns in the base EMP\_TAB table of EMP\_DEPT\_OJ1 are modifiable through the view, because EMP\_TAB is a key-preserved table in the join.

The following view also contains an outer join:

```

CREATE VIEW Emp_dept_oj2 AS
SELECT e.Empno, e.Ename, e.Deptno, d.Dname, d.Loc
FROM Emp_tab e, Dept_tab d
WHERE e.Deptno (+) = d.Deptno;

```

The statement:

```

SELECT * FROM Emp_dept_oj2;

```

Results in:

EMPNO	ENAME	DEPTNO	DNAME	LOC
7782	CLARK	10	ACCOUNTING	NEW YORK
7839	KING	10	ACCOUNTING	NEW YORK
7934	MILLER	10	ACCOUNTING	NEW YORK
7369	SMITH	20	RESEARCH	DALLAS
7876	ADAMS	20	RESEARCH	DALLAS
7902	FORD	20	RESEARCH	DALLAS
7788	SCOTT	20	RESEARCH	DALLAS
7566	JONES	20	RESEARCH	DALLAS
7499	ALLEN	30	SALES	CHICAGO
7698	BLAKE	30	SALES	CHICAGO
7654	MARTIN	30	SALES	CHICAGO
7900	JAMES	30	SALES	CHICAGO
7844	TURNER	30	SALES	CHICAGO
7521	WARD	30	SALES	CHICAGO
			OPERATIONS	BOSTON

15 rows selected.

In this view, EMP\_TAB is no longer a key-preserved table, because the EMPNO column in the result of the join can have nulls (the last row in the SELECT above). So, UPDATE, DELETE, and INSERT operations cannot be performed on this view.

In the case of views containing an outer join on other nested views, a table is key preserved if the view or views containing the table are merged into their outer views, all the way to the top. A view which is being outer-joined is currently merged only if it is "simple." For example:

```
SELECT Col1, Col2, ... FROM T;
```

The select list of the view has no expressions, and there is no WHERE clause.

Consider the following set of views:

```
CREATE VIEW Emp_v AS
  SELECT Empno, Ename, Deptno
  FROM Emp_tab;
CREATE VIEW Emp_dept_oj1 AS
  SELECT e.*, Loc, d.Dname
  FROM Emp_v e, Dept_tab d
  WHERE e.Deptno = d.Deptno (+);
```

In these examples, EMP\_V is merged into EMP\_DEPT\_OJ1 because EMP\_V is a simple view, and so EMP\_TAB is a key-preserved table. But if EMP\_V is changed as follows:

```
CREATE VIEW Emp_v_2 AS
  SELECT Empno, Ename, Deptno
  FROM Emp_tab
  WHERE Sal > 1000;
```

Then, because of the presence of the WHERE clause, EMP\_V\_2 cannot be merged into EMP\_DEPT\_OJ1, and hence EMP\_TAB is no longer a key-preserved table.

If you are in doubt whether a view is modifiable, then you can SELECT from the view USER\_UPDATABLE\_COLUMNS to see if it is. For example:

```
SELECT * FROM USER_UPDATABLE_COLUMNS WHERE TABLE_NAME = 'EMP_DEPT_VIEW';
```

This might return:

OWNER	TABLE_NAME	COLUMN_NAM	UPD
SCOTT	EMP_DEPT_V	EMPNO	NO
SCOTT	EMP_DEPT_V	ENAME	NO
SCOTT	EMP_DEPT_V	DEPTNO	NO
SCOTT	EMP_DEPT_V	DNAME	NO
SCOTT	EMP_DEPT_V	LOC	NO

5 rows selected.

## Managing Sequences

The sequence generator generates sequential numbers, which can help to generate unique primary keys automatically, and to coordinate keys across multiple rows or tables.

Without sequences, sequential values can only be produced programmatically. A new primary key value can be obtained by selecting the most recently produced value and incrementing it. This method requires a lock during the transaction and causes multiple users to wait for the next value of the primary key; this waiting is known as *serialization*. If you have such constructs in your applications, then you should replace them with access to sequences. Sequences eliminate serialization and improve the concurrency of your application.

The following sections explain how to create, alter, use, and drop sequences using SQL commands.

## Creating Sequences

Use the SQL command `CREATE SEQUENCE` to create a sequence. The following statement creates a sequence used to generate employee numbers for the `EMPNO` column of the `EMP_TAB` table:

```
CREATE SEQUENCE Emp_sequence
  INCREMENT BY 1
  START WITH 1
  NOMAXVALUE
  NOCYCLE
  CACHE 10;
```

Notice that several parameters can be specified to control the function of sequences. You can use these parameters to indicate whether the sequence is ascending or descending, the starting point of the sequence, the minimum and maximum values, and the interval between sequence values. The `NOCYCLE` option indicates that the sequence cannot generate more values after reaching its maximum or minimum value.

The `CACHE` option of the `CREATE SEQUENCE` command pre-allocates a set of sequence numbers and keeps them in memory so that they can be accessed faster. When the last of the sequence numbers in the cache have been used, another set of numbers is read into the cache.

**See Also:** For additional implications for caching sequence numbers when using the Oracle Parallel Server, see *Oracle8i Parallel Server Documentation Set: Oracle8i Parallel Server Concepts; Oracle8i Parallel Server Setup and Configuration Guide; Oracle8i Parallel Server Administration, Deployment, and Performance*.

General information about caching sequence numbers is included in "[Caching Sequence Numbers](#)" on page 2-33.

### Privileges Required to Create a Sequence

To create a sequence in your schema, you must have the `CREATE SEQUENCE` system privilege. To create a sequence in another user's schema, you must have the `CREATE ANY SEQUENCE` privilege.

## Altering Sequences

You can change any of the parameters that define how corresponding sequence numbers are generated; however, you cannot alter a sequence to change the starting number of a sequence. To do this, you must drop and re-create the sequence.

Use the SQL command `ALTER SEQUENCE` to alter a sequence. For example:

```
ALTER SEQUENCE Emp_sequence
INCREMENT BY 10
MAXVALUE 10000
CYCLE
CACHE 20;
```

### Privileges Required to Alter a Sequence

To alter a sequence, your schema must contain the sequence, or you must have the `ALTER ANY SEQUENCE` system privilege.

## Using Sequences

Once a sequence is defined, it can be accessed and incremented by multiple users with no waiting. Oracle does not wait for a transaction that has incremented a sequence to complete before that sequence can be incremented again.

The examples outlined in the following sections show how sequences can be used in master/detail table relationships. Assume an order entry system is partially comprised of two tables, `ORDERS_TAB` (master table) and `LINE_ITEMS_TAB` (detail

table), that hold information about customer orders. A sequence named `ORDER_SEQ` is defined by the following statement:

```
CREATE SEQUENCE Order_seq
  START WITH 1
  INCREMENT BY 1
  NOMAXVALUE
  NOCYCLE
  CACHE 20;
```

## Referencing a Sequence

A sequence is referenced in SQL statements with the `NEXTVAL` and `CURRVAL` pseudocolumns; each new sequence number is generated by a reference to the sequence's pseudocolumn `NEXTVAL`, while the current sequence number can be repeatedly referenced using the pseudo-column `CURRVAL`.

`NEXTVAL` and `CURRVAL` are not reserved words or keywords and can be used as pseudo-column names in SQL statements such as `SELECTS`, `INSERTS`, or `UPDATES`.

**Generating Sequence Numbers with `NEXTVAL`** To generate and use a sequence number, reference `seq_name.NEXTVAL`. For example, assume a customer places an order. The sequence number can be referenced in a values list. For example:

```
INSERT INTO Orders_tab (Orderno, Custno)
  VALUES (Order_seq.NEXTVAL, 1032);
```

Or, the sequence number can be referenced in the `SET` clause of an `UPDATE` statement. For example:

```
UPDATE Orders_tab
  SET Orderno = Order_seq.NEXTVAL
  WHERE Orderno = 10112;
```

The sequence number can also be referenced outermost `SELECT` of a query or subquery. For example:

```
SELECT Order_seq.NEXTVAL FROM dual;
```

As defined, the first reference to `ORDER_SEQ.NEXTVAL` returns the value 1. Each subsequent statement that references `ORDER_SEQ.NEXTVAL` generates the next sequence number (2, 3, 4, . . .). The pseudo-column `NEXTVAL` can be used to generate as many new sequence numbers as necessary. However, only a single sequence number can be generated per row. In other words, if `NEXTVAL` is referenced more

than once in a single statement, then the first reference generates the next number, and all subsequent references in the statement return the same number.

Once a sequence number is generated, the sequence number is available only to the session that generated the number. Independent of transactions committing or rolling back, other users referencing `ORDER_SEQ.NEXTVAL` obtain unique values. If two users are accessing the same sequence concurrently, then the sequence numbers each user receives might have gaps because sequence numbers are also being generated by the other user.

**Using Sequence Numbers with CURRVAL** To use or refer to the current sequence value of your session, reference `seq_name.CURRVAL`. `CURRVAL` can only be used if `seq_name.NEXTVAL` has been referenced in the current user session (in the current or a previous transaction). `CURRVAL` can be referenced as many times as necessary, including multiple times within the same statement. The next sequence number is not generated until `NEXTVAL` is referenced. Continuing with the previous example, you would finish placing the customer's order by inserting the line items for the order:

```
INSERT INTO Line_items_tab (Orderno, Partno, Quantity)
VALUES (Order_seq.CURRVAL, 20321, 3);
```

```
INSERT INTO Line_items_tab (Orderno, Partno, Quantity)
VALUES (Order_seq.CURRVAL, 29374, 1);
```

Assuming the `INSERT` statement given in the previous section generated a new sequence number of 347, both rows inserted by the statements in this section insert rows with order numbers of 347.

**Uses and Restrictions of NEXTVAL and CURRVAL** `CURRVAL` and `NEXTVAL` can be used in the following places:

- `VALUES` clause of `INSERT` statements
- The `SELECT` list of a `SELECT` statement
- The `SET` clause of an `UPDATE` statement

`CURRVAL` and `NEXTVAL` cannot be used in these places:

- A subquery
- A view's query or snapshot's query
- A `SELECT` statement with the `DISTINCT` operator
- A `SELECT` statement with a `GROUP BY` or `ORDER BY` clause

- A `SELECT` statement that is combined with another `SELECT` statement with the `UNION`, `INTERSECT`, or `MINUS` set operator
- The `WHERE` clause of a `SELECT` statement
- `DEFAULT` value of a column in a `CREATE TABLE` or `ALTER TABLE` statement
- The condition of a `CHECK` constraint

### Caching Sequence Numbers

Sequence numbers can be kept in the sequence cache in the System Global Area (SGA). Sequence numbers can be accessed more quickly in the sequence cache than they can be read from disk.

The sequence cache consists of entries. Each entry can hold many sequence numbers for a single sequence.

Follow these guidelines for fast access to all sequence numbers:

- Be sure the sequence cache can hold all the sequences used concurrently by your applications.
- Increase the number of values for each sequence held in the sequence cache.

**The Number of Entries in the Sequence Cache** When an application accesses a sequence in the sequence cache, the sequence numbers are read quickly. However, if an application accesses a sequence that is not in the cache, then the sequence must be read from disk to the cache before the sequence numbers are used.

If your applications use many sequences concurrently, then your sequence cache might not be large enough to hold all the sequences. In this case, access to sequence numbers might often require disk reads. For fast access to all sequences, be sure your cache has enough entries to hold all the sequences used concurrently by your applications.

The number of entries in the sequence cache is determined by the initialization parameter `SEQUENCE_CACHE_ENTRIES`. The default value for this parameter is 10 entries. Oracle creates and uses sequences internally for auditing, grants of system privileges, grants of object privileges, profiles, debugging stored procedures, and labels. Be sure your sequence cache has enough entries to hold these sequences as well as sequences used by your applications.

If the value for your `SEQUENCE_CACHE_ENTRIES` parameter is too low, then it is possible to skip sequence values. For example, assume that this parameter is set to 4, and that you currently have four cached sequences. If you create a fifth sequence,

then it will replace the least recently used sequence in the cache. All of the remaining values in this displaced sequence are lost. In other words, if the displaced sequence originally held 10 cached sequence values, and only one had been used, then nine would be lost when the sequence was displaced.

**The Number of Values in Each Sequence Cache Entry** When a sequence is read into the sequence cache, sequence values are generated and stored in a cache entry. These values can then be accessed quickly. The number of sequence values stored in the cache is determined by the `CACHE` parameter in the `CREATE SEQUENCE` statement. The default value for this parameter is 20.

This `CREATE SEQUENCE` statement creates the `SEQ2` sequence so that 50 values of the sequence are stored in the `SEQUENCE` cache:

```
CREATE SEQUENCE Seq2
  CACHE 50;
```

The first 50 values of `SEQ2` can then be read from the cache. When the 51st value is accessed, the next 50 values will be read from disk.

Choosing a high value for `CACHE` allows you to access more successive sequence numbers with fewer reads from disk to the sequence cache. However, if there is an instance failure, then all sequence values in the cache are lost. Cached sequence numbers also could be skipped after an export and import if transactions continue to access the sequence numbers while the export is running.

If you use the `NOCACHE` option in the `CREATE SEQUENCE` statement, then the values of the sequence are not stored in the sequence cache. In this case, every access to the sequence requires a disk read. Such disk reads slow access to the sequence. This `CREATE SEQUENCE` statement creates the `SEQ3` sequence so that its values are never stored in the cache:

```
CREATE SEQUENCE Seq3
  NOCACHE;
```

### Privileges Required to Use a Sequence

To use a sequence, your schema must contain the sequence or you must have been granted the `SELECT` object privilege for another user's sequence.

## Dropping Sequences

To drop a sequence, use the SQL command `DROP SEQUENCE`. For example, the following statement drops the `ORDER_SEQ` sequence:

```
DROP SEQUENCE Order_seq;
```

When you drop a sequence, its definition is removed from the data dictionary. Any synonyms for the sequence remain, but return an error when referenced.

### Privileges Required to Drop a Sequence

You can drop any sequence in your schema. To drop a sequence in another schema, you must have the `DROP ANY SEQUENCE` system privilege.

## Managing Synonyms

A synonym is an alias for a table, view, snapshot, sequence, procedure, function, or package. The following sections explain how to create, use, and drop synonyms using SQL commands.

### Creating Synonyms

Use the SQL command `CREATE SYNONYM` to create a synonym. The following statement creates a public synonym named `PUBLIC_EMP` on the `EMP_TAB` table contained in the schema of `JWARD`:

```
CREATE PUBLIC SYNONYM Public_emp FOR jward.Emp_tab;
```

### Privileges Required to Create a Synonym

You must have the `CREATE SYNONYM` system privilege to create a private synonym in your schema, or the `CREATE ANY SYNONYM` system privilege to create a private synonym in another user's schema. To create a public synonym, you must have the `CREATE PUBLIC SYNONYM` system privilege.

### Using Synonyms

A synonym can be referenced in a DML statement the same way that the underlying object of the synonym can be referenced. For example, if a synonym named `EMP_TAB` refers to a table or view, then the following statement is valid:

```
INSERT INTO Emp_tab (Empno, Ename, Job)
VALUES (Emp_sequence.NEXTVAL, 'SMITH', 'CLERK');
```

If the synonym named `FIRE_EMP` refers to a stand-alone procedure or package procedure, then you could execute it in SQL\*Plus or Enterprise Manager with the command

```
EXECUTE Fire_emp(7344);
```

You can also use synonyms for GRANT and REVOKE statements, but not with other DML statements.

### Privileges Required to Use a Synonym

You can successfully use any private synonym contained in your schema or any public synonym, assuming that you have the necessary privileges to access the underlying object, either explicitly, from an enabled role, or from PUBLIC. You can also reference any private synonym contained in another schema if you have been granted the necessary object privileges for the private synonym. You can only reference another user's synonym using the object privileges that you have been granted. For example, if you have the SELECT privilege for the JWARD.EMP\_TAB synonym, then you can query the JWARD.EMP\_TAB synonym, but you cannot insert rows using the synonym for JWARD.EMP\_TAB.

## Dropping Synonyms

To drop a synonym, use the SQL command DROP SYNONYM. To drop a private synonym, omit the PUBLIC keyword; to drop a public synonym, include the PUBLIC keyword. The following statement drops the private synonym named EMP\_TAB:

```
DROP SYNONYM Emp_tab;
```

The following statement drops the public synonym named PUBLIC\_EMP:

```
DROP PUBLIC SYNONYM Public_emp;
```

When you drop a synonym, its definition is removed from the data dictionary. All objects that reference a dropped synonym remain (for example, views and procedures) but become invalid.

### Privileges Required to Drop a Synonym

You can drop any private synonym in your own schema. To drop a private synonym in another user's schema, you must have the DROP ANY SYNONYM system privilege. To drop a public synonym, you must have the DROP PUBLIC SYNONYM system privilege.

## Creating Multiple Tables and Views in One Operation

You can create several tables and views and grant privileges in one operation using the SQL command `CREATE SCHEMA`. The `CREATE SCHEMA` command is useful if you want to guarantee the creation of several tables and views and grants in one operation; if an individual table or view creation fails or a grant fails, then the entire statement is rolled back, and none of the objects are created or the privileges granted.

For example, the following statement creates two tables and a view that joins data from the two tables:

```
CREATE SCHEMA AUTHORIZATION scott
  CREATE VIEW Sales_staff AS
    SELECT Empno, Ename, Sal, Comm
    FROM Emp_tab
    WHERE Deptno = 30 WITH CHECK OPTION CONSTRAINT
      Sales_staff_cnst

CREATE TABLE Dept_tab (
  Deptno      NUMBER(3) PRIMARY KEY,
  Dname       VARCHAR2(15),
  Loc         VARCHAR2(25))
CREATE TABLE Emp_tab (
  Empno       NUMBER(5) PRIMARY KEY,
  Ename       VARCHAR2(15) NOT NULL,
  Job         VARCHAR2(10),
  Mgr         NUMBER(5),
  Hiredate    DATE DEFAULT (sysdate),
  Sal         NUMBER(7,2),
  Comm        NUMBER(7,2),
  Deptno      NUMBER(3) NOT NULL
  CONSTRAINT Dept_fkey REFERENCES Dept_tab(Deptno))

GRANT SELECT ON Sales_staff TO human_resources;
```

The `CREATE SCHEMA` command does not support Oracle extensions to the ANSI `CREATE TABLE` and `CREATE VIEW` commands (for example, the `STORAGE` clause).

### Privileges Required to Create Multiple Schema Objects

To create schema objects, such as multiple tables, using the `CREATE SCHEMA` command, you must have the required privileges for any included operation.

## Naming Schema Objects

You should decide when you want to use partial and complete global object names in the definition of views, synonyms, and procedures. Keep in mind that database names should be stable, and databases should not be unnecessarily moved within a network.

In a distributed database system, each database should have a unique global name. The global name is composed of the database name and the network domain that contains the database. Each schema object in the database then has a global object name consisting of the schema object name and the global database name.

Because Oracle ensures that the schema object name is unique within a database, you can ensure that it is unique across all databases by assigning unique global database names. You should coordinate with your database administrator on this task, because it is usually the DBA who is responsible for assigning database names.

## Name Resolution in SQL Statements

An object name takes the following form:

```
[schema.]name[@database]
```

Some examples include:

```
Emp_tab  
Scott.Emp_tab  
Scott.Emp_tab@Personnel
```

A session is established when a user logs onto a database. Object names are resolved relative to the current user session. The username of the current user is the default schema. The database to which the user has directly logged-on is the default database.

Oracle has separate namespaces for different classes of objects. All objects in the same namespace must have distinct names, but two objects in different namespaces can have the same name. Tables, views, snapshots, sequences, synonyms, procedures, functions, and packages are in a single namespace. Triggers, indexes, and clusters each have their own individual namespace. For example, there can be a table, trigger, and index all named SCOTT.EMP\_TAB.

Based on the context of an object name, Oracle searches the appropriate namespace when resolving the name to an object. For example, in the following statement:

```
DROP CLUSTER Test
```

Oracle looks up `TEST` in the cluster namespace.

Rather than supplying an object name directly, you can also refer to an object using a synonym. A private synonym name has the same syntax as an ordinary object name. A public synonym is implicitly in the `PUBLIC` schema, but users cannot explicitly qualify a synonym with the schema `PUBLIC`.

Synonyms can only be used to reference objects in the same namespace as tables. Due to the possibility of synonyms, the following rules are used to resolve a name in a context that requires an object in the table namespace:

1. Look up the name in the table namespace.
2. If the name resolves to an object that is not a synonym, then no further work is necessary.
3. If the name resolves to a private synonym, then replace the name with the definition of the synonym and return to step 1.
4. If the name was originally qualified with a schema, then return an error; otherwise, check if the name is a public synonym.
5. If the name is not a public synonym, return an error; otherwise, then replace the name with the definition of the public synonym and return to step 1.

When global object names are used in a distributed database (either explicitly or indirectly within a synonym), the local Oracle session resolves the reference as is locally required (for example, resolving a synonym to a remote table's global object name). After the partially resolved statement is shipped to the remote database, the remote Oracle session completes the resolution of the object as above.

**See Also:** See *Oracle8i Concepts* for more information about name resolution in a distributed database.

## Renaming Schema Objects

If necessary, you can rename some schema objects using two different methods: drop and re-create the object, or rename the object using the SQL command `RENAME`.

---

---

**Note:** If you drop an object and re-create it, then all privilege grants for the object are lost when the object is dropped. Privileges must be granted again when the object is re-created.

---

---

If you use the `RENAME` command to rename a table, view, sequence, or a private synonym of a table, view, or sequence, then grants made for the object are carried forward for the new name, and the next statement renames the `SALES_STAFF` view:

```
RENAME Sales_staff TO Dept_30;
```

You cannot rename a stored PL/SQL program unit, public synonym, index, or cluster. To rename such an object, you must drop and re-create it.

Renaming a schema object has the following effects:

- All views and PL/SQL program units dependent on a renamed object become invalid (must be recompiled before next use).
- All synonyms for a renamed object return an error when used.

### Privileges Required to Rename an Object

To rename an object, you must be the owner of the object.

## Renaming the Schema

The following statement sets the current schema of the session to the schema name given in the statement.

```
ALTER SESSION SET CURRENT_SCHEMA = <schema name>
```

Any subsequent SQL statements will use this schema name for the schema qualifier when the qualifier is missing. Note that the session still has only the privileges of the current user and does not acquire any extra privileges by the above `ALTER SESSION` statement.

For example:

```
CONNECT scott/tiger
ALTER SESSION SET CURRENT_SCHEMA = joe;
SELECT * FROM emp_tab;
```

Since `emp_tab` is not schema-qualified, the table name is resolved under schema `joe`. But if `scott` does not have `select` privilege on table `joe.emp_tab`, then `scott` cannot execute the `SELECT` statement.

## Listing Information about Schema Objects

The data dictionary provides many views that provide information about schema objects. The following is a summary of the views associated with schema objects:

- `ALL_OBJECTS`, `USER_OBJECTS`
- `ALL_CATALOG`, `USER_CATALOG`
- `ALL_TABLES`, `USER_TABLES`
- `ALL_TAB_COLUMNS`, `USER_TAB_COLUMNS`
- `ALL_TAB_COMMENTS`, `USER_TAB_COMMENTS`
- `ALL_COL_COMMENTS`, `USER_COL_COMMENTS`
- `ALL_VIEWS`, `USER_VIEWS`
- `ALL_MVIEWS`, `USER_MVIEWS`
- `ALL_INDEXES`, `USER_INDEXES`
- `ALL_IND_COLUMNS`, `USER_IND_COLUMNS`
- `USER_CLUSTERS`
- `USER_CLU_COLUMNS`
- `ALL_SEQUENCES`, `USER_SEQUENCES`
- `ALL_SYNONYMS`, `USER_SYNONYMS`
- `ALL_DEPENDENCIES`, `USER_DEPENDENCIES`

**Example 1: Listing Different Schema Objects by Type** The following query lists all of the objects owned by the user issuing the query:

```
SELECT Object_name, Object_type FROM User_objects;
```

The query above might return results similar to the following:

OBJECT_NAME	OBJECT_TYPE
EMP_DEPT	CLUSTER
EMP_TAB	TABLE

```

DEPT_TAB          TABLE
EMP_DEPT_INDEX   INDEX
PUBLIC_EMP       SYNONYM
EMP_MGR          VIEW
    
```

**Example 2: Listing Column Information** Column information, such as name, datatype, length, precision, scale, and default data values, can be listed using one of the views ending with the `_COLUMNS` suffix. For example, the following query lists all of the default column values for the `EMP_TAB` and `DEPT_TAB` tables:

```

SELECT Table_name, Column_name, Data_default
      FROM User_tab_columns
      WHERE Table_name = 'DEPT_TAB' OR Table_name = 'EMP_TAB';
    
```

Considering the example statements at the beginning of this section, a display similar to the one below is displayed:

TABLE_NAME	COLUMN_NAME	DATA_DEFAULT
DEPT_TAB	DEPTNO	
DEPT_TAB	DNAME	
DEPT_TAB	LOC	('NEW YORK')
EMP_TAB	EMPNO	
EMP_TAB	ENAME	
EMP_TAB	JOB	
EMP_TAB	MGR	
EMP_TAB	HIREDATE	(sysdate)
EMP_TAB	SAL	
EMP_TAB	COMM	
EMP_TAB	DEPTNO	

---



---

**Note:** Not all columns have a user-specified default. These columns assume `NULL` when rows that do not specify values for these columns are inserted.

---



---

**Example 3: Listing Dependencies of Views and Synonyms** When you create a view or a synonym, the view or synonym is based on its underlying base object. The `_DEPENDENCIES` data dictionary views can be used to reveal the dependencies for a view and the `_SYNONYMS` data dictionary views can be used to list the base object of a synonym. For example, the following query lists the base objects for the synonyms created by the user JWARD:

```
SELECT Table_owner, Table_name
       FROM All_synonyms
       WHERE Owner = 'JWARD';
```

This query could return information similar to the following:

TABLE_OWNER	TABLE_NAME
SCOTT	DEPT_TAB
SCOTT	EMP_TAB



---

## Selecting a Datatype

This chapter discusses how to use Oracle *built-in* datatypes in applications. Topics include:

- [Oracle Built-In Datatypes](#)
- [ANSI/ISO, DB2, and SQL/DS Datatypes](#)
- [Data Conversion](#)

**See Also:** For information about more complex types, such as object types, varrays, and nested tables, refer to *Oracle8i Application Developer's Guide - Object-Relational Features* .

## Oracle Built-In Datatypes

A datatype associates a fixed set of properties with the values that can be used in a column of a table or in an argument of a procedure or function. These properties cause Oracle to treat values of one datatype differently from values of another datatype. For example, Oracle can add values of `NUMBER` datatype, but not values of `RAW` datatype.

Oracle supplies the following built-in datatypes:

- Character datatypes
  - `CHAR`
  - `NCHAR`
  - `VARCHAR2` and `VARCHAR`
  - `NVARCHAR2`
  - `CLOB`
  - `NCLOB`
  - `LONG`
- `NUMBER` datatype
- `DATE` datatype
- Binary datatypes
  - `BLOB`
  - `BFILE`
  - `RAW`
  - `LONG RAW`

Another datatype, `ROWID`, is used for values in the `ROWID` pseudocolumn, which represents the unique address of each row in a table.

**See Also:** See *Oracle Call Interface Programmer's Guide* for general descriptions of these datatypes, and see *Oracle8i Application Developer's Guide - Large Objects (LOBs)* for information about the `LOB` datatypes.

[Table 3-1](#) summarizes the information about each Oracle built-in datatype.

**Table 3–1 Summary of Oracle Built-In Datatypes**

<b>Datatype</b>	<b>Description</b>	<b>Column Length and Default</b>
CHAR ( <i>size</i> )	Fixed-length character data of length <i>size</i> bytes	Fixed for every row in the table (with trailing blanks); maximum size is 2000 bytes per row, default size is 1 byte per row. Consider the character set (one-byte or multibyte) before setting <i>size</i> .
VARCHAR2 ( <i>size</i> )	Variable-length character data	Variable for each row, up to 4000 bytes per row: Consider the character set (one-byte or multibyte) before setting <i>size</i> : A maximum <i>size</i> must be specified.
NCHAR ( <i>size</i> )	Fixed-length character data of length <i>size</i> characters or bytes, depending on the national character set	Fixed for every row in the table (with trailing blanks). Column <i>size</i> is the number of characters for a fixed-width national character set or the number of bytes for a varying-width national character set. Maximum <i>size</i> is determined by the number of bytes required to store one character, with an upper limit of 2000 bytes per row. Default is 1 character or 1 byte, depending on the character set.
NVARCHAR2 ( <i>size</i> )	Variable-length character data of length <i>size</i> characters or bytes, depending on national character set: A maximum <i>size</i> must be specified	Variable for each row. Column <i>size</i> is the number of characters for a fixed-width national character set or the number of bytes for a varying-width national character set. Maximum <i>size</i> is determined by the number of bytes required to store one character, with an upper limit of 4000 bytes per row. Default is 1 character or 1 byte, depending on the character set.
CLOB	Single-byte character data	Up to $2^{32} - 1$ bytes, or 4 gigabytes.

**Table 3–1 Summary of Oracle Built-In Datatypes (Cont.)**

NCLOB	Single-byte or fixed-length multibyte national character set (NCHAR) data	Up to $2^{32} - 1$ bytes, or 4 gigabytes.
LONG	Variable-length character data	Variable for each row in the table, up to $2^{31} - 1$ bytes, or 2 gigabytes, per row. Provided for backward compatibility.
NUMBER ( <i>p</i> , <i>s</i> )	Variable-length numeric data.: Maximum precision <i>p</i> and/or scale <i>s</i> is 38	Variable for each row. The maximum space required for a given column is 21 bytes per row.
DATE	Fixed-length date and time data, ranging from Jan. 1, 4712 B.C.E. to Dec. 31, 4712 C.E.	Fixed at 7 bytes for each row in the table. Default format is a string (such as DD-MON-RR) specified by NLS_DATE_FORMAT parameter.
BLOB	Unstructured binary data	Up to $2^{32} - 1$ bytes, or 4 gigabytes.
BFILE	Binary data stored in an external file	Up to $2^{32} - 1$ bytes, or 4 gigabytes.
RAW ( <i>size</i> )	Variable-length raw binary data	Variable for each row in the table, up to 2000 bytes per row. A maximum <i>size</i> must be specified. Provided for backward compatibility.
LONG RAW	Variable-length raw binary data	Variable for each row in the table, up to $2^{31} - 1$ bytes, or 2 gigabytes, per row. Provided for backward compatibility.
ROWID	Binary data representing row addresses	Fixed at 10 bytes (extended ROWID) or 6 bytes (restricted ROWID) for each row in the table.

## Using Character Datatypes

Use the character datatypes to store alphanumeric data.

- `CHAR` and `NCHAR` datatypes store fixed-length character strings.
- `VARCHAR2` and `NVARCHAR2` datatypes store variable-length character strings. (The `VARCHAR` datatype is synonymous with the `VARCHAR2` datatype.)
- `CLOB` and `NCLOB` datatypes store single-byte and multibyte character strings of up to four gigabytes.

**See Also:** *Oracle8i Application Developer's Guide - Large Objects (LOBs)*

- The `LONG` datatype stores variable-length character strings containing up to two gigabytes, but with many restrictions.

**See Also:** ["Restrictions on LONG and LONG RAW Data"](#)

This datatype is provided for backward compatibility with existing applications; in general, new applications should use `CLOB` and `NCLOB` datatypes to store large amounts of character data.

When deciding which datatype to use for a column that will store alphanumeric data in a table, consider the following points of distinction:

### Space Usage

- To store data more efficiently, use the `VARCHAR2` datatype. The `CHAR` datatype blank-pads and stores trailing blanks up to a fixed column length for all column values, while the `VARCHAR2` datatype does not blank-pad or store trailing blanks for column values.

### Comparison Semantics

- Use the `CHAR` datatype when you require ANSI compatibility in comparison semantics (when trailing blanks are not important in string comparisons). Use the `VARCHAR2` when trailing blanks are important in string comparisons.

### Future Compatibility

- The `CHAR` and `VARCHAR2` datatypes are and will always be fully supported. At this time, the `VARCHAR` datatype automatically corresponds to the `VARCHAR2` datatype and is reserved for future use.

CHAR, VARCHAR2, and LONG data is automatically converted from the database character set to the character set defined for the user session by the NLS\_LANGUAGE parameter, where these are different.

### Column Lengths for Single-Byte and Multibyte Character Sets

The lengths of CHAR and VARCHAR2 columns are specified in bytes rather than characters, and are constrained as such. The lengths of NCHAR and NVARCHAR2 columns are specified either in bytes or in characters, depending on the national character set being used.

When using a multibyte database character encoding scheme, consider carefully the space required for tables with character columns. If the database character encoding scheme is single-byte, then the number of bytes and the number of characters in a column is the same. If it is multibyte, then there generally is no such correspondence. A character might consist of one or more bytes depending upon the specific multibyte encoding scheme, and whether shift-in/shift-out control codes are present.

#### See Also:

- *Oracle8i National Language Support Guide*
- *Oracle8i SQL Reference*
- *Oracle8i Time Series User's Guide*

for information about National Language Support features of Oracle and support for different character encoding schemes.

### Comparison Semantics

Oracle compares CHAR and NCHAR values using *blank-padded comparison semantics*. If two values have different lengths, then Oracle adds blanks at the end of the shorter value, until the two values are the same length. Oracle then compares the values character-by-character up to the first character that differs. The value with the greater character in the first differing position is considered greater. Two values that differ only in the number of trailing blanks are considered equal.

Oracle compares VARCHAR2 and NVARCHAR2 values using *non-padded comparison semantics*. Two values are considered equal only if they have the same characters and are of equal length. Oracle compares the values character-by-character up to the first character that differs. The value with the greater character in that position is considered greater.

Because Oracle blank-pads values stored in CHAR columns but not in VARCHAR2 columns, a value stored in a VARCHAR2 column may take up less space than if it

were stored in a `CHAR` column. For this reason, a full table scan on a large table containing `VARCHAR2` columns may read fewer data blocks than a full table scan on a table containing the same data stored in `CHAR` columns. If your application often performs full table scans on large tables containing character data, then you might be able to improve performance by storing this data in `VARCHAR2` columns rather than in `CHAR` columns.

However, performance is not the only factor to consider when deciding which of these datatypes to use. Oracle uses different semantics to compare values of each datatype. You might choose one datatype over the other if your application is sensitive to the differences between these semantics. For example, if you want Oracle to ignore trailing blanks when comparing character values, then you must store these values in `CHAR` columns.

**See Also:** For more information on comparison semantics for these datatypes, see the *Oracle8i Reference*.

## Using the NUMBER Datatype

Use the `NUMBER` datatype to store real numbers in a fixed-point or floating-point format. Numbers using this datatype are guaranteed to be portable among different Oracle platforms, and offer up to 38 decimal digits of precision. You can store positive and negative numbers of magnitude  $1 \times 10^{-130}$  to  $9.99... \times 10^{125}$ , as well as zero, in a `NUMBER` column.

For numeric columns you can specify the column as a floating-point number:

```
Column_name NUMBER
```

Or, you can specify a precision (total number of digits) and scale (number of digits to right of decimal point):

```
Column_name NUMBER (<precision>, <scale>)
```

Although not required, specifying the precision and scale for numeric fields provides extra integrity checking on input. If a precision is not specified, then the column stores values as given. Table 3–2 shows examples of how data would be stored using different scale factors.

**Table 3–2** How Scale Factors Affect Numeric Data Storage

Input Data	Stored As	Specified As
7,456,123.89	NUMBER	7456123.89

**Table 3–2 How Scale Factors Affect Numeric Data Storage**

Input Data	Stored As	Specified As
7,456,123.89	NUMBER (9)	7456124
7,456,123.89	NUMBER (9,2)	7456123.89
7,456,123.89	NUMBER (9,1)	7456123.9
7,456,123.89	NUMBER (6)	(not accepted, exceeds precision)
7,456,123.89	NUMBER (7, -2)	7456100

**See Also:** For information about the internal format for the NUMBER datatype, see *Oracle8i Concepts*.

## Using the DATE Datatype

Use the DATE datatype to store point-in-time values (dates and times) in a table. The DATE datatype stores the century, year, month, day, hours, minutes, and seconds.

Oracle uses its own internal format to store dates. Date data is stored in fixed-length fields of seven bytes each, corresponding to century, year, month, day, hour, minute, and second.

**See Also:** See the *Oracle Call Interface Programmer's Guide* for a complete description of the Oracle internal date format.

### Date Format

For input and output of dates, the standard Oracle default date format is DD-MON-RR. For example:

```
'13-NOV-1992'
```

To change this default date format on an instance-wide basis, use the NLS\_DATE\_FORMAT parameter. To change the format during a session, use the ALTER SESSION statement. To enter dates that are not in the current default date format, use the TO\_DATE function with a format mask. For example:

```
TO_DATE ('November 13, 1992', 'MONTH DD, YYYY')
```

**See Also:** Oracle Julian dates might not be compatible with Julian dates generated by other date algorithms. For information about Julian dates, see *Oracle8i Concepts*.

If the date format `DD-MON-YY` is used, then `YY` indicates the year in the 20th century (for example, `31-DEC-92` is December 31, 1992). If you want to indicate years in any century other than the 20th century, then use a different format mask, such as the default `RR`.

### Time Format

Time is stored in 24-hour format `#HH:MM:SS`. By default, the time in a date field is 12:00:00 A.M. (midnight) if no time portion is entered. In a time-only entry, the date portion defaults to the first day of the current month. To enter the time portion of a date, use the `TO_DATE` function with a format mask indicating the time portion, as in:

---



---

**Note:** You may need to set up the following data structures for certain examples to work:

```
CREATE TABLE Birthdays_tab (Bname VARCHAR2(20),Bday DATE)
```

---



---

```
INSERT INTO Birthdays_tab (bname, bday) VALUES
('ANNIE',TO_DATE('13-NOV-92 10:56 A.M.','DD-MON-YY HH:MI A.M.'));
```

To compare dates that have time data, use the SQL function `TRUNC` if you want to ignore the time component. Use the SQL function `SYSDATE` to return the system date and time. The `FIXED_DATE` initialization parameter allows you to set `SYSDATE` to a constant; this can be useful for testing.

## Establishing Year 2000 Compliance

An application must satisfy the following criteria to meet the requirements for Year 2000 (Y2K) compliance:

- Process date information before, during, and after 1st January 2000 without error. This entails accepting date input, providing date output, storing date information and performing calculation on dates or portions of dates.
- Provide services as published in its documentation before, during and after 1st January 2000 without changes in operation resulting from the advent of the new century.

- Respond to two digit date input in a way that resolves ambiguity as to the century in a clearly defined manner.
- Manage the leap year occurring in the year 2000 according to the quad-centennial rule.

These criteria are a superset of the Year 2000 conformance requirements set out by the British Standards Institute in DISC PD-2000-1 A Definition of Year 2000 Conformity Requirements.

You can warrant your application as Y2K compliant only if you have validated its conformance at all three of the following system levels:

- Hardware
- System software, including databases, transaction processors and operating systems
- Application software, from third parties or developed in-house

### **Oracle Server Year 2000 Compliance**

The Oracle Server is Year 2000 compliant. No operational problems are expected with the Oracle Server, networking and system management products. Oracle's Development Organization has conducted tests of various Year 2000 operational scenarios to verify that there is no impact to users at the turn of the century. These scenarios included tests of replication, point-in-time recovery, distributed transactions. System management and networking features across time zones / datelines / centuries have also been tested.

Please note that Oracle's Year 2000 product compliance does not eliminate the need for you to test your own applications. Most importantly, your application software has to be tested on the Oracle Server to ensure that operations having to do with the year 2000 perform as promised. This test is critical even if the application software is certified to be Year 2000 compliant because there are no universal protocol definitions that can guarantee conformance without such testing.

### **Centuries and the Year 2000**

Oracle stores year data with the century information. For example, the Oracle database stores 1996 or 2001, and not just 96 or 01. The `DATE` datatype always stores a four-digit year internally, and all other dates stored internally in the database have four digit years. Oracle utilities such as `import`, `export`, and `recovery` also deal properly with four-digit years.

Applications that use the Oracle RDBMS (Oracle7 and Oracle8 Server) and exploit the `DATE` data type (for date and/or date with time values) need have no concerns about their stored data when the year 2000 approaches. The Oracle7 and Oracle8 Server `DATE` data type stores date and time data to a precision that includes a four digit year and a time component down to seconds (typically 'YYYY:MM:DD:HH24:MI:SS')

However, some applications might be written with an assumption about the year (such as assuming that everything is 19xx). The application might hand over a two-digit year to the database, and the procedures that Oracle uses for determining the century could be different from what the programmer expects (see "[Programming Hints and Tips](#)" on page 3-15). For this reason, you should review and test your code with regard to the Year 2000.

### The 'RR' Date Format

The `RR` date format element of the `TO_DATE` and `TO_CHAR` functions allows a database site to default the century to different values depending on the two-digit year, so that years 50 to 99 default to 19xx and years 00 to 49 default to 20xx. Therefore, regardless of the current century at the time the data is entered, the 'RR' format will ensure that the year stored in the database is as follows:

- If the current year is in the second half of the century (50 - 99), and a two-digit year between '00' and '49' is entered, this will be stored as a 'next century' year. For example, '02' entered in 1996 will be stored as '2002'.
- If the current year is in the second half of the century (50 - 99), and a two-digit year between '50' and '99' is entered, this will be stored as a 'current century' year. For example, '97' entered in 1996 will be stored as '1997'.
- If the current year is in the first half of the century (00 - 49), and a two-digit year between '00' and '49' is entered, this will be stored as a 'current century' year. For example, '02' entered in 2001 will be stored as '2002'.
- If the current year is in the first half of the century (00 - 49), and a two-digit year between '50' and '99' is entered, this will be stored as a 'previous century' year. For example, '97' entered in 2001 will be stored as '1997'.

The 'RR' date format is available for inserting and updating `DATE` data in the database. It is not required for retrieval or query of data already stored in the database as Oracle has always stored the `YEAR` component of a date in its four-digit form.

Here is an example of the `RR` usage:

```
INSERT INTO emp (empno, deptno, hiredate) VALUES
(9999, 20, TO_DATE('01-jan-03', 'DD-MON-RR'));
```

```
INSERT INTO emp (empno, deptno, hiredate) VALUES
(8888, 20, TO_DATE('01-jan-67', 'DD-MON-RR'));
```

```
SELECT empno, deptno,
       TO_CHAR(hiredate, 'DD-MON-YYYY') hiredate
FROM emp;
```

This produces the following data:

EMPNO	DEPTNO	HIREDATE
8888	20	01-JAN-1967
9999	20	01-JAN-2003

### The 'CC' Date Format

The CC date format element of the TO\_CHAR function returns the century of a given date. An example of CC usage follows:

```
SELECT TO_CHAR(TO_DATE('01-JAN-2000', 'DD-MON-YYYY'), 'CC') CC FROM DUAL;
```

This produces the following result:

```
CC
----
20
```

A second example of CC usage follows:

```
SELECT TO_CHAR(TO_DATE('01-JAN-2001', 'DD-MON-YYYY'), 'CC') CC FROM DUAL;
```

This produces the following result:

```
CC
----
21
```

The CC date format element of the TO\_CHAR function sets the century value to one greater than the first two digits of a four-digit year (for example, '20' from '1900'). For years that are a multiple of 100, this is not the true century. Strictly speaking, the century of '1900' is not the twentieth century (which began in 1901) but rather the nineteenth century.

The following workaround computes the correct century for any Common Era (CE, formerly known as AD) date. If *userdate* is a CE date for which you want the true century, use the following expression:

```
SELECT DECODE (TO_CHAR (Hiredate, 'YY'),
              '00', TO_CHAR (Hiredate - 366, 'CC'),
              TO_CHAR (Hiredate, 'CC')) FROM Emp_tab;
```

This expression works as follows: Get the last two digits of the year. If it is '00', then it is a year in which the Oracle century is one year too large, and compute a date in the preceding year (whose Oracle century is the desired true century). Otherwise, use the Oracle century.

**See Also:** For more information about date format codes, see *Oracle8 SQL Reference*.

### Storing Dates in Character Data Types

Where applications store date values in CHAR or VARCHAR2 datatypes, and the century information is not maintained, you will need to modify the application to include routines which ensure that such dates are treated appropriately when affected by the change in century. You can do this by changing the strings to maintain century information or, with certain constraints, by using the 'RR' date format when interpreting the string as a date.

If you are creating a new application, or if you are modifying an application to ensure that dates stored as character strings are Year 2000 compliant, we advise that you convert dates to use the Oracle DATE data type. If this is not feasible, store the dates in a form which is language and format independent, and which handles full years. For example, utilize 'YYYY/MM/DD' plus the time element as 'HH24:MI:SS' if necessary. Note that dates stored in this form must be converted to the correct external format whenever they are displayed or received from users or other programs.

The format 'YYYY/MM/DD HH24:MI:SS' has the following advantages:

- It is language-independent in that the months are numeric.
- It contains the full four-digit year so centuries are unambiguous.
- The time is represented fully. Since the most significant elements occur first, character-based sort operations will process the dates correctly.

The “S” format element prefixes BC dates with “-“.

### Viewing Date Settings

The following views will enable you to verify what your settings are:

- `V$NLS_DATABASE_PARAMETERS` — shows instance wide NLS parameters whether explicitly declared in the `INIT.ORA` or defaulting.
- `NLS_SESSION_PARAMETERS` — shows current session values which may have been changed by means of `ALTER SESSION`

A format model is a character that describes the format of `DATE` or `NUMBER` data stored in a character string. You may use the format model as an argument of the `TO_CHAR` or `TO_DATE` function for one of the following:

- To specify the format for Oracle to use in returning a value from the database.
- To specify the format for a value you have specified for Oracle to store in the database.

Please note that the format does not change the internal representation of the value in the database.

### Altering Date Settings

You may set the date format in your environment or as the default for the entire database. If you set this in your environment it will override the setting in the initialization parameter.

Change the `NLS_DATE_FORMAT` parameter settings in the following order:

1. Set the Client side e.g., Windows NT registry and Unix environment variable
2. Set the Session using `ALTER SESSION SET NLS_DATE_FORMAT`. To change the date format for the session, issue the following SQL command:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-RR'
```

3. Set the Server using the `init.ora` `NLS_DATE_FORMAT` parameter. To change the default date format for the entire database, change `INIT.ORA` to include the following

```
NLS_DATE_FORMAT = DD-MON-RR
```

The `NLS_DATE_FORMAT` setting relies on the above order. Therefore, for a client/server application, `NLS_DATE_FORMAT` needs to

be set on the server and the client.

---



---

**Caution:** Note that changing this parameter at the *database* level will change all existing date fields as described above. We suggest you make changes at the *session* level unless all users and all currently running applications process dates in the range 1950-2049.

---



---

## Programming Hints and Tips

In this section we describe some common programming problems around Y2K compliance. These problems may seem to derive from incorrect Year 2000 processing by the database engine, but on closer inspection are seen to arise from incorrect use of Oracle technology.

### Example 1

Your application may have defined the year of a date using a column of `CHAR(2)` or `NUMBER(2)` in order to save disk space. This can lead to unpredictable results when 20xx dates are mixed with 19xx dates. To resolve this, modify your application to use the full 4-digit year.

### Example 2

Your application may be designed to store a 4-digit year, but the code may allow for the incorrect storage of 2-digit year rows with the 4-digit year rows. This will lead to unpredictable results for queries by date if the date columns contains dates earlier than 1900. To deal with this problem, have your application check for rows which contain dates earlier than 1900, and then adjust for this.

### Example 3

Examine your applications to determine if it processes dates prior to 1950 or later than 2049, and store the year as 2-digits. If both conditions are met, your application should not use the 'RR' format but should instead expand the 2 digit year 'YY' into a 4 digit year 'YYYY', and store the 4 digit number in the database.

### Example 4

The following unusual error helps illuminate the interaction between `NLS_DATE_FORMAT` and the Oracle 'RR' format mask. The following is a syntactically correct statement but contains a logical flaw:

```
SELECT TO_CHAR(TO_DATE(LAST_DAY('01-FEB-00'), 'DD-MON-RR'), 'MM/DD/RRRR')
```

```
FROM DUAL;
```

The above query will return 02/28/2000. This is consistent with the defined behavior of the 'RR' format element. However, since the year 2000 is a leap year, this is incorrect.

The problem is that the operation is using the default `NLS_DATE_FORMAT`, which is 'DD-MON-YY'. If the `NLS_DATE_FORMAT` is changed to 'DD-MON-RR', then the same select returns 02/29/2000, which is the correct value.

Let us evaluate the query as the Oracle Server engine does. The first function processed is the innermost function, `LAST_DAY`. Because `NLS_DATE_FORMAT` is `YY`, this correctly returns 2/28, because it is using the year 1900 to evaluate the expression. The value 2/28 is then returned to the next outer function. So, the `TO_DATE` and `TO_CHAR` functions format the value 02/28/00 using the 'RR' format mask and display the result as 02/28/2000.

If `SELECT LAST_DAY('01-FEB-00') FROM DUAL` is issued, the result will change depending on the `NLS_DATE_FORMAT`. With 'YY', the `LAST_DAY` returned is 28-Feb-00 because the year is interpreted as 1900. With 'RR', the `LAST_DAY` returned is 29-Feb-00 because the year is interpreted as 2000. The year 1900 is not a leap year whereas the year 2000 is a leap year.

### Example 5

When the `DECODE` function is used and if the third argument has data type `CHAR`, `VARCHAR2`, or if it is `NULL`, then Oracle converts the return value to datatype `VARCHAR2`. Therefore, the following statement:

```
INSERT INTO destination_table (date_column)
  SELECT DECODE('31.12.2000', '00000000', NULL,
    TO_DATE('31.12.2000', 'DD.MM.YYYY'))
  FROM DUAL;
```

inserts date 31.12.1900.

Another sample statement:

```
INSERT INTO destination_table (date_column)
  SELECT DECODE('01.11.1999', '00000000', NULL, sysdate+1000)
  FROM DUAL;
```

inserts date 04.10.1901.

In the above examples, the third argument in the `DECODE` argument list is a `NULL` value, so Oracle implicitly converted the `DATE` value to a `VARCHAR2` string using

the default format mask. This is DD-MON-YY, hence loses the first two digits of the year.

*Note: When inserting the record into a table, Oracle implicitly converts the string into a date, using the first 2-digits of the current year. To ensure the correct year is interpreted, set NLS\_DATE\_FORMAT using 'RR' or 'YYYY'.*

### Example 6

If creating a partitioned table using a DATE data type column in the partition key, use a 4-digit year when specifying date ranges. For example:

```
CREATE TABLE stock_xactions (stock_symbol CHAR(5),
    stock_series CHAR(1),
    num_shares NUMBER(10),
    price NUMBER(5,2),
    trade_date DATE)
STORAGE (INITIAL 100K NEXT 50K) LOGGING
PARTITION BY RANGE (trade_date)
    (PARTITION sx1992 VALUES LESS THAN (TO_DATE('01-JAN-1993','DD-MON-YYYY'))
TABLESPACE ts0
    NOLOGGING,
    PARTITION sx1993 VALUES LESS THAN (TO_DATE('01-JAN-1994','DD-MON-YYYY'))
TABLESPACE ts1,
    PARTITION sx1994 VALUES LESS THAN (TO_DATE('01-JAN-1995','DD-MON-YYYY'))
TABLESPACE ts2);
```

### Example 7

Oracle views depend on the session state. In particular, a predicate with a 2-digit year, such as, "where col '12-MAY-99' ", is allowed in a view. Interpretation of the full 4-digit year depends on the setting of NLS\_DATE\_FORMAT.

## Using the LONG Datatype

---

---

**Note:** The LONG datatype is provided for backward compatibility with existing applications. For new applications, you should use the CLOB and NCLOB datatypes for large amounts of character data. See *Oracle8i Application Developer's Guide - Large Objects (LOBs)* for information about the CLOB and NCLOB datatypes.

---

---

The LONG datatype can store variable-length character data containing up to two gigabytes of information. The length of LONG values might be limited by the memory available on your computer.

You can use columns defined as LONG in SELECT lists, SET clauses of UPDATE statements, and VALUES clauses of INSERT statements. LONG columns have many of the characteristics of VARCHAR2 columns.

### Restrictions on LONG and LONG RAW Data

Although LONG (and LONG RAW; see below) columns have many uses, their use has some restrictions:

- Only one LONG column is allowed per table.
- LONG columns cannot be indexed.
- LONG columns cannot appear in integrity constraints.
- LONG columns cannot be used in WHERE , GROUP BY , ORDER BY, or CONNECT BY clauses or with the DISTINCT operator in SELECT statements.
- LONG columns cannot be referenced by SQL functions (such as SUBSTR or INSTR).
- LONG columns cannot be used in the SELECT list of a subquery or queries combined by the set operators UNION, INTERSECT, or MINUS.
- LONG columns cannot be used in SQL expressions.
- LONG columns cannot be referenced when creating a table with a query (CREATE TABLE... AS SELECT...) or when inserting into a table or view with a query (INSERT INTO... SELECT...).
- A variable or argument of a PL/SQL program unit cannot be declared using the LONG datatype.

- Variables in database triggers cannot be declared using the `LONG` or `LONG RAW` datatypes.
- References to `:NEW` and `:OLD` in database triggers cannot be used with `LONG` or `LONG RAW` columns.
- `LONG` and `LONG RAW` columns cannot be used in distributed SQL statements.
- `LONG` and `LONG RAW` columns cannot be replicated.

**Note:** If you design tables containing `LONG` or `LONG RAW` data, then you should place each `LONG` or `LONG RAW` column in a table separate from any other data associated with it, rather than storing the `LONG` or `LONG RAW` column and its associated data together in the same table. You can then relate the two tables with a referential integrity constraint. This design allows SQL statements that access only the associated data to avoid reading through `LONG` or `LONG RAW` data.

### Example of LONG Datatype

To store information on magazine articles, including the texts of each article, create two tables. For example:

```
CREATE TABLE Article_header
  (Id          NUMBER PRIMARY KEY,
   Title       VARCHAR2(200),
   First_author VARCHAR2(30),
   Journal     VARCHAR2(50),
   Pub_date   DATE);

CREATE TABLE article_text
  (Id          NUMBER
   REFERENCES Article_header,
   Text       LONG);
```

The `ARTICLE_TEXT` table stores only the text of each article. The `ARTICLE_HEADER` table stores all other information about the article, including the title, first author, and journal and date of publication. The two tables are related by the referential integrity constraint on the `ID` column of each table.

This design allows SQL statements to query data other than the text of an article without reading through the text. If you want to select all first authors published in Nature magazine during July 1991, then you can issue this statement that queries the `ARTICLE_HEADER` table:

```
SELECT First_author
   FROM Article_header
  WHERE Journal = 'NATURE'
     AND TO_CHAR(Pub_date, 'MM YYYY') = '07 1991';
```

If the text of each article were stored in the same table with the first author, publication, and publication date, then Oracle would need to read through the text to perform this query.

## Using RAW and LONG RAW Datatypes

---

---

**Note:** The RAW and LONG RAW datatypes are provided for backward compatibility with existing applications. For new applications, you should use the BLOB and BFILE datatypes for large amounts of binary data.

---

---

**See Also:** See *Oracle8i Application Developer's Guide - Large Objects (LOBs)* for information about the BLOB and BFILE datatypes.

The RAW and LONG RAW datatypes store data that is not interpreted by Oracle (that is, not converted when moving data between different systems). These datatypes are intended for binary data and byte strings. For example, LONG RAW can store graphics, sound, documents, and arrays of binary data; the interpretation is dependent on the use.

Net8 and the Export and Import utilities do not perform character conversion when transmitting RAW or LONG RAW data. When Oracle automatically converts RAW or LONG RAW data to and from CHAR data (as is the case when entering RAW data as a literal in an INSERT statement), the data is represented as one hexadecimal character representing the bit pattern for every four bits of RAW data. For example, one byte of RAW data with bits 11001011 is displayed and entered as 'CB'.

LONG RAW data cannot be indexed, but RAW data can be indexed.

**See Also:** For more information about restrictions on LONG RAW data, see "[Restrictions on LONG and LONG RAW Data](#)".

## ROWIDs and the ROWID Datatype

Every row in a nonclustered table of an Oracle database is assigned a unique ROWID that corresponds to the physical address of a row's row piece (initial row piece if the row is chained among multiple row pieces). In the case of clustered tables, rows in different tables that are in the same data block can have the same ROWID.

Each table in an Oracle database internally has a pseudocolumn named ROWID.

**See Also:** *Oracle8i Concepts* for general information about the ROWID pseudocolumn and the ROWID datatype.

### Extended ROWID Format

The Oracle Server uses an *extended ROWID* format, which supports features such as table partitions, index partitions, and clusters.

The extended ROWID includes the following information:

- Data object (segment) identifier
- Datafile identifier
- Block identifier
- Row identifier

The data object identifier is an identification number that Oracle assigns to schema objects in the database, such as nonpartitioned tables or partitions. For example:

```
SELECT DATA_OBJECT_ID FROM ALL_OBJECTS
       WHERE OWNER = 'SCOTT' AND OBJECT_NAME = 'EMP_TAB';
```

This query returns the data object identifier for the EMP\_TAB table in the SCOTT schema.

**See Also:** *Oracle8i Supplied PL/SQL Packages Reference* for information about other ways to get the data object identifier, using the DBMS\_ROWID package functions.

### Different Forms of the ROWID

Oracle documentation uses the term ROWID in different ways, depending on context. These uses are explained in this section.

**Internal ROWID** The internal ROWID format is an internal structure which holds information that the server code needs to access a row. The restricted internal ROWID is 6 bytes on most platforms; the extended ROWID is 10 bytes on these platforms.

**ROWID Pseudocolumn** Each table and nonjoined view has a pseudocolumn called ROWID. For example:

```
CREATE TABLE T_tab (col1 Rowid);
INSERT INTO T_tab SELECT Rowid FROM Emp_tab WHERE Empno = 7499;
```

This command returns the ROWID pseudocolumn of the row of the EMP\_TAB table that satisfies the query, and inserts it into the T1 table.

**External Character ROWID** The extended ROWID pseudocolumn is returned to the client in the form of an 18-character string (for example, "AAAA8mAALAAAAQkAAA"), which represents a base 64 encoding of the components of the extended ROWID in a four-piece format, OOOOOFFBBBBBBRRR:

- OOOOOO: The **data object number** identifies the database segment (AAAA8m in the example). Schema objects in the same segment, such as a cluster of tables, have the same data object number.
- FFF: The **datafile** that contains the row (file AAL in the example). File numbers are unique within a database.
- BBBBBB: The **data block** that contains the row (block AAAAQk in the example). Block numbers are relative to their datafile, *not* tablespace. Therefore, two rows with identical block numbers could reside in two different datafiles of the same tablespace.
- RRR: The **row** in the block (row AAA in the example).

There is no need to decode the external ROWID; you can use the functions in the DBMS\_ROWID package to obtain the individual components of the extended ROWID.

**See Also:** *Oracle8i Supplied PL/SQL Packages Reference* for information about the DBMS\_ROWID package.

The restricted ROWID pseudocolumn is returned to the client in the form of an 18-character string with a hexadecimal encoding of the datablock, row, and datafile components of the ROWID.

**External Binary ROWID** Some client applications use a binary form of the ROWID. For example, OCI and some precompiler applications can map the ROWID to a 3GL structure on bind or define calls. The size of the binary ROWID is the same for extended and restricted ROWIDs. The information for the extended ROWID is included in an unused field of the restricted ROWID structure.

The format of the extended binary ROWID, expressed as a C struct, is:

```
struct riddef {
    ub4    ridobjnum; /* data obj#--this field is
                       unused in restricted ROWIDs */
    ub2    ridfilenum;
    ub1    filler;
    ub4    ridblocknum;
    ub2    ridslotnum;
}
```

## ROWID Migration and Compatibility Issues

For backward compatibility, the restricted form of the ROWID is still supported. These ROWIDs exist in massive amounts of Oracle7 data, and the extended form of the ROWID is required only in global indexes on partitioned tables. New tables always get extended ROWIDs.

**See Also:** *Oracle8i Administrator's Guide.*

It is possible for an Oracle7 client to access an Oracle8 database. Similarly, an Oracle8 client can access an Oracle7 Server. A client in this sense can include a remote database accessing a server using database links, as well as a client 3GL or 4GL application accessing a server.

**See Also:** There is more information on the ROWID\_TO\_EXTENDED function in *Oracle8i Supplied PL/SQL Packages Reference* and *Oracle8i Migration*.

**Accessing an Oracle7 Database from an Oracle8 Client** The ROWID values that are returned are always restricted ROWIDs. Also, Oracle8 uses restricted ROWIDs when returning a ROWID value to an Oracle7 or earlier server.

The following ROWID functionality works when accessing an Oracle7 Server:

- Selecting a ROWID and using the obtained value in a WHERE clause
- WHERE CURRENT OF cursor operations

- Storing ROWIDs in user columns of ROWID or CHAR type
- Interpreting ROWIDs using the hexadecimal encoding (not recommended, use the DBMS\_ROWID functions)

**Accessing an Oracle8 Database from an Oracle7 Client** Oracle8 returns ROWIDs in the extended format. This means that you can only:

- Select a ROWID and use it in a WHERE clause
- Use WHERE CURRENT OF cursor operations
- Store ROWIDs in user columns of CHAR(18) datatype

**Import and Export** It is not possible for an Oracle7 client to import an Oracle8 table that has a ROWID column (not the ROWID pseudocolumn), if any row of the table contains an extended ROWID value.

## ANSI/ISO, DB2, and SQL/DS Datatypes

You can define columns of tables in an Oracle database using ANSI/ISO, DB2, and SQL/DS datatypes. Oracle internally converts such datatypes to Oracle datatypes.

The ANSI datatype conversions to Oracle datatypes are shown in [Table 3–3](#). The ANSI/ISO datatypes NUMERIC, DECIMAL, and DEC can specify only fixed-point numbers. For these datatypes, s defaults to 0.

**Table 3–3 ANSI Datatype Conversions to Oracle Datatypes**

ANSI SQL Datatype	Oracle Datatype
CHARACTER (n), CHAR (n)	CHAR (n)
NUMERIC (p,s), DECIMAL (p,s), DEC (p,s)	NUMBER (p,s)
INTEGER, INT, SMALLINT	NUMBER (38)
FLOAT (p)	FLOAT (p)
REAL	FLOAT (63)
DOUBLE PRECISION	FLOAT (126)
CHARACTER VARYING(n), CHAR VARYING(n)	VARCHAR2 (n)

The IBM products SQL/DS, and DB2 datatypes TIME, TIMESTAMP, GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC have no corresponding Oracle datatype and

cannot be used. The `TIME` and `TIMESTAMP` datatypes are subcomponents of the Oracle datatype `DATE`.

**Table 3–4** shows the DB2 and SQL/DS conversions.

**Table 3–4 SQL/DS, DB2 Datatype Conversions to Oracle Datatypes**

DB2 or SQL/DS Datatype	Oracle Datatype
<code>CHARACTER (n)</code>	<code>CHAR (n)</code>
<code>VARCHAR (n)</code>	<code>VARCHAR2 (n)</code>
<code>LONG VARCHAR</code>	<code>LONG</code>
<code>DECIMAL (p,s)</code>	<code>NUMBER (p,s)</code>
<code>INTEGER, SMALLINT</code>	<code>NUMBER (38)</code>
<code>FLOAT (p)</code>	<code>FLOAT (p)</code>
<code>DATE</code>	<code>DATE</code>

## Data Conversion

In some cases, Oracle allows data of one datatype where it expects data of a different datatype. Generally, an expression cannot contain values with different datatypes. However, Oracle can use the following functions to automatically convert data to the expected datatype:

- `TO_NUMBER()`
- `TO_CHAR()`
- `TO_DATE()`
- `HEXTORAW()`
- `RAWTOHEX()`
- `ROWIDTOCHAR()`
- `CHARTOROWID()`

Implicit datatype conversions work according to the rules explained below.

### Rule 1: Assignments

For assignments, Oracle can automatically convert the following:

- VARCHAR2 or CHAR to NUMBER
- NUMBER to VARCHAR2
- VARCHAR2 or CHAR to DATE
- DATE to VARCHAR2
- VARCHAR2 or CHAR to ROWID
- ROWID to VARCHAR2
- VARCHAR2 or CHAR to HEX
- HEX to VARCHAR2

The assignment succeeds if Oracle can convert the datatype of the value used in the assignment to that of the assignment's target.

For the examples in the following list, assume a package with a public variable and a table declared as in the following statements:

---

---

**Note:** You may need to set up the following data structures for certain examples to work:

```
CREATE PACKAGE Test_Pack AS var1 CHAR(5); END;  
CREATE TABLE Table1_tab (col1 NUMBER);
```

---

---

- `variable := expression`

The datatype of *expression* must be either the same as, or convertible to, the datatype of *variable*. For example, Oracle automatically converts the data provided in the following assignment within the body of a stored procedure:

```
VAR1 := 0;
```

- `INSERT INTO table VALUES (expression1, expression2, ...)`

The datatypes of *expression1*, *expression2*, and so on, must be either the same as, or convertible to, the datatypes of the corresponding columns in *table*. For example, Oracle automatically converts the data provided in the following INSERT statement for TABLE1 (see table definition above):

```
INSERT INTO Table1_tab VALUES ('19');
```

- `UPDATE table SET column = expression`

The datatype of *expression* must be either the same as, or convertible to, the datatype of *column*. For example, Oracle automatically converts the data provided in the following UPDATE statement issued against TABLE1:

```
UPDATE Table1_tab SET col1 = '30';
```

- SELECT *column* INTO *variable* FROM *table*

The datatype of *column* must be either the same as, or convertible to, the datatype of *variable*. For example, Oracle automatically converts data selected from the table before assigning it to the variable in the following statement:

```
SELECT Col1 INTO Var1 FROM Table1_tab WHERE Col1 = 30;
```

## Rule 2: Expression Evaluation

---



---

**Caution:** You may need to set up data structures for certain examples to work:

---



---

For expression evaluation, Oracle can automatically perform the same conversions as for assignments. An expression is converted to a type based on its context. For example, operands to arithmetic operators are converted to NUMBER and operands to string functions are converted to VARCHAR2.

Oracle can automatically convert the following:

- VARCHAR2 or CHAR to NUMBER
- VARCHAR2 or CHAR to DATE

Character to NUMBER conversions succeed only if the character string represents a valid number. Character to DATE conversions succeed only if the character string satisfies the session default format, which is specified by the initialization parameter NLS\_DATE\_FORMAT.

Some common types of expressions follow:

- Simple expressions, such as:

```
Comm + '500'
```

- Boolean expressions, such as:

```
Bonus > Sal / '10'
```

- Function and procedure calls, such as:

```
MOD (Counter, '2')
```

- WHERE clause conditions, such as:

```
WHERE Hiredate = TO_DATE('1997-01-01', 'yyyy-mm-dd')
```

- WHERE clause conditions, such as:

```
WHERE Rowid = 'AAAAaoAATAAADAAA'
```

In general, Oracle uses the rule for expression evaluation when a datatype conversion is needed in places not covered by the rule for assignment conversions.

In assignments of the form:

```
variable := expression
```

Oracle first evaluates *expression* using the conversions covered by Rule 2; *expression* can be as simple or complex as desired. If it succeeds, then the evaluation of *expression* results in a single value and datatype. Then, Oracle tries to assign this value to the assignment's target using Rule 1.

---

# Maintaining Data Integrity

This chapter explains how to enforce the business rules associated with your database and prevent the entry of invalid information into tables by using integrity constraints. Topics include the following:

- [Using Integrity Constraints](#)
- [Using Referential Integrity Constraints](#)
- [Referential Integrity in a Distributed Database](#)
- [Using CHECK Integrity Constraints](#)
- [Defining Integrity Constraints](#)
- [Enabling and Disabling Integrity Constraints](#)
- [Altering Integrity Constraints](#)
- [Dropping Integrity Constraints](#)
- [Managing FOREIGN KEY Integrity Constraints](#)
- [Listing Integrity Constraint Definitions](#)

## Using Integrity Constraints

You can define integrity constraints to enforce business rules on data in your tables. Once an integrity constraint is enabled, all data in the table must conform to the rule that it specifies. If you subsequently issue a SQL statement that modifies data in the table, then Oracle ensures that the resulting data satisfies the integrity constraint. Without integrity constraints, such business rules must be enforced programmatically by your application.

## When to Enforce Business Rules with Integrity Constraints

Enforcing rules with integrity constraints is more reliable than enforcing the equivalent rules by issuing SQL statements in your application. The semantics of integrity constraints are very clearly defined, so the internal operations that Oracle performs to enforce them are optimized beneath the level of SQL statements in Oracle. Because your applications use SQL, they cannot achieve this level of optimization.

**Example** To ensure that each employee in the EMP\_TAB table works for a department that is listed in the DEPT\_TAB table, first create a PRIMARY KEY constraint on the DEPTNO column of the DEPT\_TAB table with the following statement:

```
ALTER TABLE Dept_tab
  ADD PRIMARY KEY (Deptno);
```

Then create a referential integrity constraint on the DEPTNO column of the EMP\_TAB table that references the primary key of the DEPT\_TAB table. For example:

```
ALTER TABLE Emp_tab
  ADD FOREIGN KEY (Deptno) REFERENCES Dept_tab(Deptno);
```

If you subsequently add a new employee record to the table, then Oracle automatically ensures that its department number appears in the department table.

To enforce this rule without integrity constraints, you can use a trigger to test each new employee record to ensure that its department number belongs to an existing department. This testing involves issuing a SELECT statement to query the DEPT\_TAB table. Because SELECT in Oracle uses "consistent read", the query might miss uncommitted changes from other transactions. Integrity constraints avoid this problem.

## When to Enforce Business Rules in Applications

In some cases, you might want to enforce business rules through your application as well as through integrity constraints. Enforcing a business rule in your application might provide faster feedback to the user than an integrity constraint. For example, if your application accepts 20 values from the user and then issues an `INSERT` statement containing these values, then you might want your user to be notified immediately after entering a value that violates a business rule.

Because integrity constraints are enforced only when a SQL statement is issued, an integrity constraint can only notify the user of a bad value after the user has entered all 20 values and the application has issued the `INSERT` statement. However, you can design your application to verify the integrity of each value as it is entered, and notify the user immediately in the event of a bad value.

## Creating Indexes for Use with Constraints

All enabled unique and primary keys require indexes, and foreign keys should almost always be indexed. Although unique and primary keys can create unique indexes for you, when you need an index for performance reasons, you should not rely on an index that is automatically created for key columns. Instead, create the index first, by hand.

Note that:

- Constraints use existing indexes; they do not create indexes unless necessary.
- Unique and primary keys can use non-unique as well as unique indexes. They can even use just the first few columns of non-unique indexes.
- At most one unique or primary key can use each non-unique index.
- The column orders in the index and the constraint do not need to match.
- If you need to check whether an index is used by a constraint, for example when you want to drop the index, the object number of the index used by a unique or primary key constraint is stored in `CDEF$.ENABLED` for that constraint. It is not shown in any catalog view.

## Using NOT NULL Integrity Constraints

By default, all columns can contain nulls. Only define `NOT NULL` constraints for columns of a table that absolutely require values at all times.

For example, in the `EMP_TAB` table, an employee's manager or hire date might be temporarily omitted. Some employees might not have a commission. Therefore,

these three columns should not have `NOT NULL` integrity constraints. However, an employee name might be required in each row, making this column a good candidate for a `NOT NULL` integrity constraint.

`NOT NULL` constraints are often combined with other types of integrity constraints to further restrict the values that can exist in specific columns of a table. Use the combination of `NOT NULL` and `UNIQUE` key integrity constraints to force the input of values in the `UNIQUE` key; this combination of data integrity rules eliminates the possibility that any new row's data will ever attempt to conflict with an existing row's data.

Because Oracle indexes do not store keys that are all null, if you want to allow index-only scans of the table or some other operation that requires indexing all rows, put a `NOT NULL` constraint on at least one indexed column.

**See Also:** ["Relationships Between Parent and Child Tables"](#) on page 4-10

**Figure 4-1 NOT NULL Integrity Constraints**

The constraint on `ENAME` is specified by: `ALTER TABLE "EMP" MODIFY "ENAME" NOT NULL`

Table EMP							
EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7329	SMITH	CEO		17-DEC-85	9,000.00		20
7499	ALLEN	VP-SALES	7329	20-FEB-90	7,500.00	100.00	30
7521	WARD	MANAGER	7499	22-FEB-90	5,000.00	200.00	30
7566	JONES	SALESMAN	7521	02-APR-90	2,975.00	400.00	30

**NOT NULL Constraint**  
(no row may contain a null value for this column)

**Absence of NOT NULL Constraint**  
(any row can contain a null for this column)

## Setting Default Column Values

Legal default values include any literal, or any expression that does not refer to a sequence, PL/SQL function, column, `LEVEL`, `ROWNUM`, or `PRIOR`. Default values can include the expressions `SYSDATE`, `USER`, `USERENV`, and `UID`. The datatype of the default literal or expression must match or be convertible to the column datatype.

If you do not explicitly define a default value for a column, the default for the column is implicitly set to `NULL`.

### When to Use Default Values

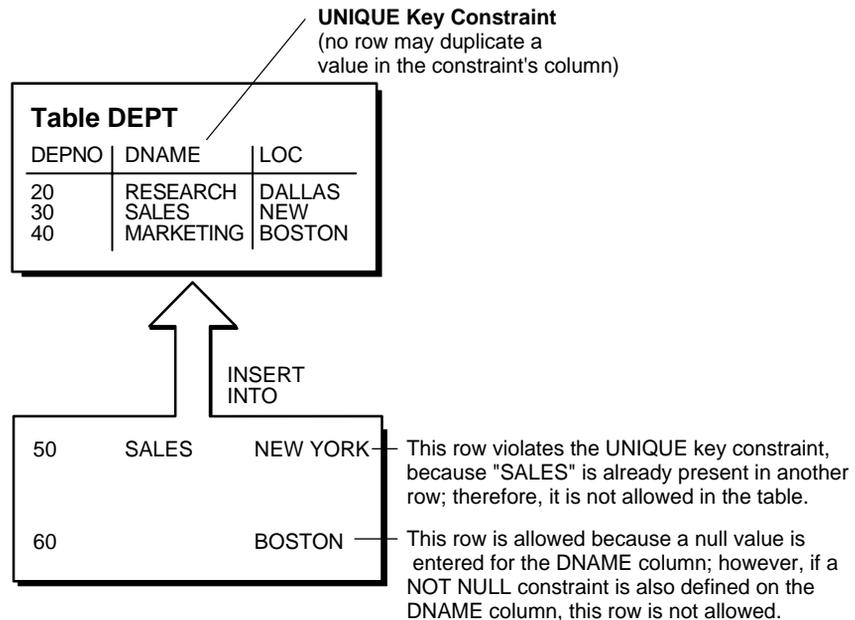
Only assign default values to columns that contain a typical value. For example, in the `DEPT_TAB` table, if most departments are located at one site, then the default value for the `LOC` column can be set to this value (such as `NEW YORK`).

Defaults are also useful when you use a view to make a subset of a table's columns visible. For example, you might allow users to insert rows into a table through a view. The view is defined to show all columns pertinent to end-user operations; however, the base table might also have a column named `INSERTER`, not included in the definition of the view, which logs the user that originally inserts each row of the table. The column named `INSERTER` can record the name of the user that inserts a row by defining the column with the `USER` function. For example:

```
. . . , inserter VARCHAR2(30) DEFAULT USER, . . .
```

**See Also:** For another example of assigning a default column value, refer to the section "Creating Tables".

Figure 4–2 A UNIQUE Key Constraint



## Choosing a Table's Primary Key

Each table can have one primary key. A primary key allows each row in a table to be uniquely identified and ensures that no duplicate rows exist. Use the following guidelines when selecting a primary key:

- Whenever practical, use a column containing a sequence number. It is a simple way to satisfy all the other guidelines.
- Choose a column whose data values are unique.

The purpose of a table's primary key is to uniquely identify each row of the table. Therefore, the column or set of columns in the primary key must contain unique values for each row.

- Choose a column whose data values are never changed.

A primary key value is only used to identify a row in the table; primary key values should never contain any data that is used for any other purpose. Therefore, primary key values should rarely need to be changed.

- Choose a column that does not contain any nulls.  
A `PRIMARY KEY` constraint, by definition, does not allow the input of any row with a null in any column that is part of the primary key.
- Choose a column that is short and numeric.  
Short primary keys are easy to type. You can use sequence numbers to easily generate numeric primary keys.
- Avoid choosing composite primary keys.  
Although composite primary keys are allowed, they do not satisfy the previous recommendations. For example, composite primary key values are long and cannot be assigned by sequence numbers.

## Using UNIQUE Key Integrity Constraints

Choose unique keys carefully. In many situations, unique keys are incorrectly comprised of columns that should be part of the table's primary key (see the previous section for more information about primary keys). When deciding whether to use a `UNIQUE` key constraint, use the rule that a `UNIQUE` key constraint is only required to prevent the duplication of the key values within the rows of the table. The data in a unique key is such that it cannot be duplicated in the table.

---

---

**Note:** Although `UNIQUE` key constraints allow the input of nulls, because of the search mechanism for `UNIQUE` constraints on more than one column, you cannot have identical values in the non-null columns of a partially null composite `UNIQUE` key constraint.

---

---

Do not confuse the concept of a unique key with that of a primary key. Primary keys are used to identify each row of the table uniquely. Therefore, unique keys should not have the purpose of identifying rows in the table.

Some examples of good unique keys include

- An employee's social security number (the primary key is the employee number)
- A truck's license plate number (the primary key is the truck number)
- A customer's phone number, consisting of the two columns `AREA` and `PHONE` (the primary key is the customer number)
- A department's name and location (the primary key is the department number)

## Using Referential Integrity Constraints

Whenever two tables are related by a common column (or set of columns), define a `PRIMARY` or `UNIQUE` key constraint on the column in the parent table, and define a `FOREIGN KEY` constraint on the column in the child table, to maintain the relationship between the two tables.

**See Also:** Depending on this relationship, you may want to define additional integrity constraints including the foreign key, as listed in the section "[Relationships Between Parent and Child Tables](#)" on page 4-10.

Figure 4-3 shows a foreign key defined on the `DEPTNO` column of the `EMP_TAB` table. It guarantees that every value in this column must match a value in the primary key of the `DEPT_TAB` table (the `DEPTNO` column); therefore, no erroneous department numbers can exist in the `DEPTNO` column of the `EMP_TAB` table.

Foreign keys can be comprised of multiple columns. However, a composite foreign key must reference a composite primary or unique key of the exact same structure (the same number of columns and datatypes). Because composite primary and unique keys are limited to 32 columns, a composite foreign key is also limited to 32 columns.

## Nulls and Foreign Keys

Foreign keys allow key values that are all null, even if there are no matching `PRIMARY` or `UNIQUE` keys.

By default (without any `NOT NULL` or `CHECK` clauses), and in accordance with the ANSI/ISO standard, the `FOREIGN KEY` constraint enforces the "match none" rule for composite foreign keys. The "full" and "partial" rules can also be enforced by using `CHECK` and `NOT NULL` constraints, as follows:

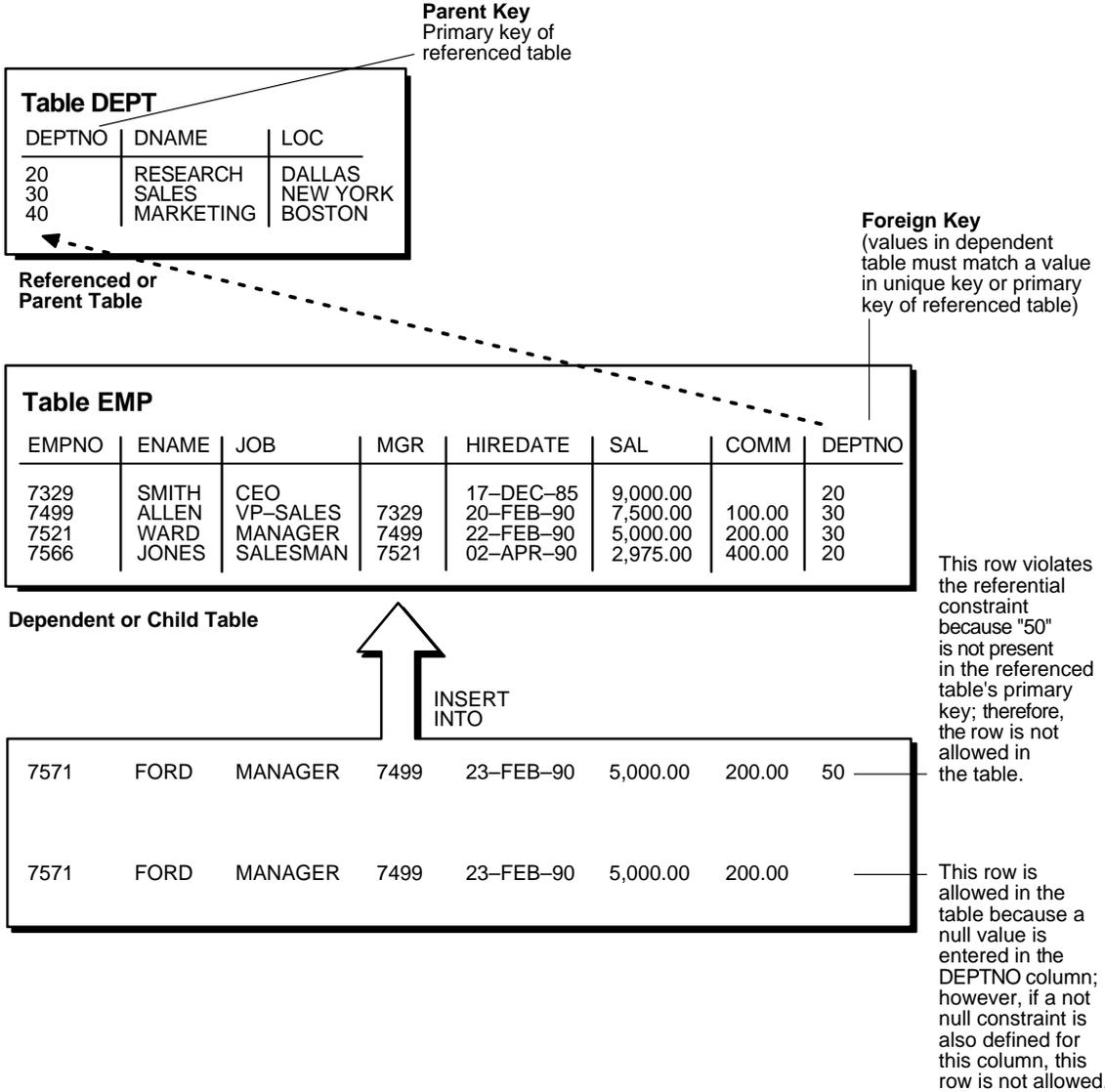
- To enforce the "match full" rule for nulls in composite foreign keys, which requires that all components of the key be null or all be non-null, define a `CHECK` constraint that allows only all nulls or all non-nulls in the composite foreign key as follows, assuming a composite key comprised of columns A, B, and C:

```
CHECK ((A IS NULL AND B IS NULL AND C IS NULL) OR
       (A IS NOT NULL AND B IS NOT NULL AND C IS NOT NULL))
```

- In general, it is not possible to use declarative referential integrity to enforce the "match partial" rule for nulls in composite foreign keys, which requires the non-null portions of the key to appear in the corresponding portions in the

primary or unique key of a single row in the referenced table. You can often use triggers to handle this case, as described in [Chapter 12, "Using Triggers"](#).

Figure 4-3 Referential Integrity Constraints



## Relationships Between Parent and Child Tables

Several relationships between parent and child tables can be determined by the other types of integrity constraints defined on the foreign key in the child table.

**No Constraints on the Foreign Key** When no other constraints are defined on the foreign key, any number of rows in the child table can reference the same parent key value. This model allows nulls in the foreign key.

This model establishes a "one-to-many" relationship between the parent and foreign keys that allows undetermined values (nulls) in the foreign key. An example of such a relationship is shown in Figure 4-3 on page 8 between `EMP_TAB` and `DEPT_TAB`; each department (parent key) has many employees (foreign key), and some employees might not be in a department (nulls in the foreign key).

**NOT NULL Constraint on the Foreign Key** When nulls are not allowed in a foreign key, each row in the child table must explicitly reference a value in the parent key because nulls are not allowed in the foreign key. However, any number of rows in the child table can reference the same parent key value.

This model establishes a "one-to-many" relationship between the parent and foreign keys. However, each row in the child table must have a reference to a parent key value; the absence of a value (a null) in the foreign key is not allowed. The same example in the previous section can be used to illustrate such a relationship. However, in this case, employees must have a reference to a specific department.

**UNIQUE Constraint on the Foreign Key** When a `UNIQUE` constraint is defined on the foreign key, one row in the child table can reference a parent key value. This model allows nulls in the foreign key.

This model establishes a "one-to-one" relationship between the parent and foreign keys that allows undetermined values (nulls) in the foreign key. For example, assume that the `EMP_TAB` table had a column named `MEMBERNO`, referring to an employee's membership number in the company's insurance plan. Also, a table named `INSURANCE` has a primary key named `MEMBERNO`, and other columns of the table keep respective information relating to an employee's insurance policy. The `MEMBERNO` in the `EMP_TAB` table should be both a foreign key and a unique key:

- To enforce referential integrity rules between the `EMP_TAB` and `INSURANCE` tables (the `FOREIGN KEY` constraint)
- To guarantee that each employee has a unique membership number (the `UNIQUE` key constraint)

**UNIQUE and NOT NULL Constraints on the Foreign Key** When both `UNIQUE` and `NOT NULL` constraints are defined on the foreign key, only one row in the child table can reference a parent key value. Because nulls are not allowed in the foreign key, each row in the child table must explicitly reference a value in the parent key.

This model establishes a "one-to-one" relationship between the parent and foreign keys that does not allow undetermined values (nulls) in the foreign key. If you expand the previous example by adding a `NOT NULL` constraint on the `MEMBERNO` column of the `EMP_TAB` table, in addition to guaranteeing that each employee has a unique membership number, then you also ensure that no undetermined values (nulls) are allowed in the `MEMBERNO` column of the `EMP_TAB` table.

## Multiple FOREIGN KEY Constraints

Oracle allows a column to be referenced by multiple `FOREIGN KEY` constraints; effectively, there is no limit on the number of dependent keys. This situation might be present if a single column is part of two different composite foreign keys.

## Deferring Constraint Checks

When Oracle checks a constraint, it signals an error if the constraint is not satisfied. You can *defer* checking the validity of constraints until the end of a transaction.

When you issue the `SET CONSTRAINTS` statement, the `SET CONSTRAINTS` mode lasts for the duration of the transaction, or until another `SET CONSTRAINTS` statement resets the mode.

---

---

**Note:** You cannot issue a `SET CONSTRAINT` statement inside a trigger.

---

---

**See Also:** For more details about the `SET CONSTRAINTS` statement, see the *Oracle8i SQL Reference*.

For general information about constraints, see *Oracle8i Concepts*.

## How To Defer Constraint Checks

**Select Appropriate Data** You may wish to defer constraint checks on `UNIQUE` and `FOREIGN` keys if the data you are working with has any of the following characteristics:

- tables are snapshots

- tables that contain a large amount of data being manipulated by another application, which may or may not return the data in the same order
- update cascade operations on FOREIGN keys

When dealing with bulk data being manipulated by outside applications, you can defer checking constraints for validity until the end of a transaction.

**Ensure Constraints Are Created Deferrable** After you have identified and selected the appropriate tables, make sure the tables' FOREIGN, UNIQUE and PRIMARY key constraints are created deferrable. You can do so by issuing a statement similar to the following:

```
CREATE TABLE dept (
    deptno NUMBER PRIMARY KEY,
    dname VARCHAR2 (30)
);
CREATE TABLE emp (
    empno NUMBER,
    ename VARCHAR2 (30),
    deptno NUMBER REFERENCES (dept),
    CONSTRAINT epk PRIMARY KEY (empno) DEFERRABLE,
    CONSTRAINT efk FOREIGN KEY (deptno)
    REFERENCES (dept. deptno) DEFERRABLE);
INSERT INTO dept VALUES (10, 'Accounting');
INSERT INTO dept VALUES (20, 'SALES');
INSERT INTO emp VALUES (1, 'Corleone', 10);
INSERT INTO emp VALUES (2, 'Costanza', 20);
COMMIT;

SET CONSTRAINT efk DEFERRED;
UPDATE dept SET deptno = deptno + 10
    WHERE deptno = 20;

SELECT * from emp ORDER BY deptno;
EMPNO      ENAME          DEPTNO
-----  -
1         Corleone         10
2         Costanza         20
UPDATE emp SET deptno = deptno + 10
    WHERE deptno = 20;
SELECT * FROM emp ORDER BY deptno;

EMPNO      ENAME          DEPTNO
-----  -
1         Corleone         10
2         Costanza         30
COMMIT;
```

**Set All Constraints Deferred** Within the application being used to manipulate the data, you must set all constraints deferred before you actually begin processing any data. Use the following DML statement to set all deferrable constraints deferred:

```
SET CONSTRAINTS ALL DEFERRED;
```

---

---

**Note:** The SET CONSTRAINTS statement applies only to the current transaction. The defaults specified when you create a constraint remain as long as the constraint exists. The ALTER SESSION SET CONSTRAINTS statement applies for the current session only.

---

---

**Check the Commit (Optional)** You can check for constraint violations before committing by issuing the SET CONSTRAINTS ALL IMMEDIATE statement just before issuing the COMMIT. If there are any problems with a constraint, this statement will fail and the constraint causing the error will be identified. If you commit while constraints are violated, the transaction will be rolled back and you will receive an error message.

## Managing Constraints That Have Associated Indexes

When you create a UNIQUE or PRIMARY key, Oracle checks to see if an existing index can be used to enforce uniqueness for the constraint. If there is no such index, Oracle creates one.

When Oracle is using a unique index to enforce a constraint, and constraints associated with the unique index are dropped or disabled, the index is dropped.

While enabled foreign keys reference a PRIMARY or UNIQUE key, you cannot disable or drop the PRIMARY or UNIQUE key constraint or the index.

---

---

**Note:** Deferrable UNIQUE and PRIMARY keys all must use non-unique indexes.

---

---

## Concurrency Control, Indexes, and Foreign Keys

You should almost always index foreign keys. The only exception is when the matching unique or primary key is never updated or deleted.

**See Also:** *Oracle8i Concepts* for information on locking mechanisms involving indexes and keys.

## Referential Integrity in a Distributed Database

Oracle does not permit declarative referential integrity constraints to be defined across nodes of a distributed database (in other words, a declarative referential integrity constraint on one table cannot specify a foreign key that references a primary or unique key of a remote table).

However, parent/child table relationships across nodes can be maintained using triggers.

**See Also:** For more information about triggers that enforce referential integrity, refer to [Chapter 12, "Using Triggers"](#).

---

---

**Note:** If you decide to define referential integrity across the nodes of a distributed database using triggers, then be aware that network failures can limit the accessibility of not only the parent table, but also the child table. For example, assume that the child table is in the `SALES` database, and the parent table is in the `HQ` database.

If the network connection between the two databases fails, then some DML statements against the child table (those that insert rows into the child table or update a foreign key value in the child table) cannot proceed, because the referential integrity triggers must have access to the parent table in the `HQ` database.

---

---

## Using CHECK Integrity Constraints

Use `CHECK` constraints when you need to enforce integrity rules based on logical expressions, such as comparisons. Never use `CHECK` constraints when any of the other types of integrity constraints can provide the necessary checking.

**See Also:** ["CHECK and NOT NULL Integrity Constraints"](#) on page 4-16

Examples of `CHECK` constraints include the following:

- A `CHECK` constraint on the `SAL` column of the `EMP_TAB` table so that no salary value is greater than 10000
- A `CHECK` constraint on the `LOC` column of the `DEPT_TAB` table so that only the locations "BOSTON", "NEW YORK", and "DALLAS" are allowed
- A `CHECK` constraint on the `SAL` and `COMM` columns to compare the `SAL` and `COMM` values of a row and prevent the `COMM` value from being greater than the `SAL` value

## Restrictions on CHECK Constraints

A `CHECK` integrity constraint requires that a condition be true or unknown for every row of the table. If a statement causes the condition to evaluate to false, then the statement is rolled back. The condition of a `CHECK` constraint has the following limitations:

- The condition must be a Boolean expression that can be evaluated using the values in the row being inserted or updated.
- The condition cannot contain subqueries or sequences.
- The condition cannot include the `SYSDATE`, `UID`, `USER`, or `USERENV SQL` functions.
- The condition cannot contain the pseudocolumns `LEVEL`, `PRIOR`, or `ROWNUM`;

**See Also:** *Oracle8i SQL Reference* for an explanation of these pseudocolumns.

- The condition cannot contain a user-defined SQL function.

## Designing CHECK Constraints

When using `CHECK` constraints, consider the ANSI/ISO standard which states that a `CHECK` constraint is violated only if the condition evaluates to false; true and unknown values do not violate a check condition. Therefore, make sure that any `CHECK` constraint that you define actually enforces the rule you need enforced.

For example, consider the following `CHECK` constraint:

```
CHECK (Sal > 0 OR Comm >= 0)
```

At first glance, this rule may be interpreted as "do not allow a row in the `EMP_TAB` table unless the employee's salary is greater than zero or the employee's commission is greater than or equal to zero." However, note that if a row is inserted with a null salary and a negative commission, then the row does not violate the `CHECK` constraint, because the entire check condition is evaluated as unknown. In this particular case, you can account for such violations by placing `NOT NULL` integrity constraints on both the `SAL` and `COMM` columns.

---

---

**Note:** If you are not sure when unknown values result in `NULL` conditions, review the truth tables for the logical operators `AND` and `OR` in *Oracle8i SQL Reference*

---

---

## Multiple CHECK Constraints

A single column can have multiple `CHECK` constraints that reference the column in its definition. There is no limit to the number of `CHECK` constraints that can be defined that reference a column.

The order in which the constraints are evaluated is not defined, so be careful not to rely on the order or to define multiple constraints that conflict with each other.

## CHECK and NOT NULL Integrity Constraints

According to the ANSI/ISO standard, a `NOT NULL` integrity constraint is an example of a `CHECK` integrity constraint, where the condition is the following:

```
CHECK (Column_name IS NOT NULL)
```

Therefore, `NOT NULL` integrity constraints for a single column can, in practice, be written in two forms: using the `NOT NULL` constraint or a `CHECK` constraint. For ease of use, you should always choose to define `NOT NULL` integrity constraints, instead of `CHECK` constraints with the `IS NOT NULL` condition.

In the case where a composite key can allow only all nulls or all values, you must use a `CHECK` integrity constraint. For example, the following expression of a `CHECK` integrity constraint allows a key value in the composite key made up of columns `C1` and `C2` to contain either all nulls or all values:

```
CHECK ((C1 IS NULL AND C2 IS NULL) OR
       (C1 IS NOT NULL AND C2 IS NOT NULL))
```

## Defining Integrity Constraints

Here are some examples showing how to create simple constraints during the prototype phase of your database design.

Notice how all constraints are given a name. Naming the constraints prevents the database from creating multiple copies of the same constraint, with different system-generated names, if the DDL is run multiple times.

**See Also:** *Oracle8i Administrator's Guide* for information on creating and maintaining constraints for a large production database.

## The CREATE TABLE Command

The following examples of `CREATE TABLE` statements show the definition of several integrity constraints:

```
CREATE TABLE Dept_tab (
  Deptno NUMBER(3) CONSTRAINT Dept_pkey PRIMARY KEY,
  Dname  VARCHAR2(15),
  Loc    VARCHAR2(15),
          CONSTRAINT Dname_ukey UNIQUE (Dname, Loc),
          CONSTRAINT Loc_check1
          CHECK (loc IN ('NEW YORK', 'BOSTON', 'CHICAGO')));
```

```
CREATE TABLE Emp_tab (
  Empno   NUMBER(5) CONSTRAINT Emp_pkey PRIMARY KEY,
  Ename   VARCHAR2(15) NOT NULL,
  Job     VARCHAR2(10),
  Mgr     NUMBER(5) CONSTRAINT Mgr_fkey
          REFERENCES Emp_tab,
  Hiredate DATE,
  Sal     NUMBER(7,2),
  Comm    NUMBER(5,2),
  Deptno  NUMBER(3) NOT NULL
```

```
CONSTRAINT dept_fkey REFERENCES Dept_tab ON DELETE CASCADE);
```

### The ALTER TABLE Command

You can also define integrity constraints using the constraint clause of the ALTER TABLE command. For example, the following examples of ALTER TABLE statements show the definition of several integrity constraints:

```
CREATE UNIQUE INDEX I_dept ON Dept_tab(deptno);
ALTER TABLE Dept_tab
    ADD CONSTRAINT Dept_pkey PRIMARY KEY (deptno);

ALTER TABLE Emp_tab
    ADD CONSTRAINT Dept_fkey FOREIGN KEY (Deptno) REFERENCES Dept_tab;
ALTER TABLE Emp_tab MODIFY (Ename VARCHAR2(15) NOT NULL);
```

You cannot create a validated constraint on a table if the table already contains any rows that would violate the constraint.

### Required Privileges

The creator of a constraint must have the ability to create tables (the CREATE TABLE or CREATE ANY TABLE system privilege), or the ability to alter the table (the ALTER object privilege for the table or the ALTER ANY TABLE system privilege) with the constraint. Additionally, UNIQUE and PRIMARY KEY integrity constraints require that the owner of the table have either a quota for the tablespace that contains the associated index or the UNLIMITED TABLESPACE system privilege. FOREIGN KEY integrity constraints also require some additional privileges.

**See Also:** ["Privileges Required for FOREIGN KEY Integrity Constraints"](#) on page 4-25

### Naming Integrity Constraints

Assign names to NOT NULL, UNIQUE KEY, PRIMARY KEY, FOREIGN KEY, and CHECK constraints using the CONSTRAINT option of the constraint clause. This name must be unique with respect to other constraints that you own. If you do not specify a constraint name, then one is assigned by Oracle.

Picking your own name makes error messages for constraint violations more understandable, and prevents the creation of multiple constraints if the SQL statements are run more than once.

See the previous examples of the `CREATE TABLE` and `ALTER TABLE` statements for examples of the `CONSTRAINT` option of the `constraint` clause. Note that the name of each constraint is included with other information about the constraint in the data dictionary.

**See Also:** "[Listing Integrity Constraint Definitions](#)" on page 4-27 for examples of data dictionary views.

#### Enabling and Disabling Integrity Constraints

This section explains the mechanisms and procedures for manually enabling and disabling integrity constraints.

enabled constraint	When a constraint is enabled, the rule defined by the constraint is enforced on the data values in the columns that define the constraint. The definition of the constraint is stored in the data dictionary.
disabled constraint	When a constraint is disabled, the rule defined by the constraint is not enforced on the data values in the columns included in the constraint; however, the definition of the constraint is retained in the data dictionary.

In summary, an integrity constraint can be thought of as a statement about the data in a database. This statement is always true when the constraint is enabled; however, the statement may or may not be true when the constraint is disabled because data in violation of the integrity constraint can be in the database.

## Why Disable Constraints?

To enforce the rules defined by integrity constraints, the constraints should always be enabled; however, in certain situations, it is desirable to disable the integrity constraints of a table temporarily for performance reasons. For example:

- When loading large amounts of data into a table using `SQL*Loader`
- When performing batch operations that make massive changes to a table (such as changing everyone's employee number by adding 1000 to the existing number)
- When importing or exporting one table at a time

In cases such as these, integrity constraints may be temporarily turned off to improve the performance of the operation.

## Integrity Constraint Violations

If a row of a table does not adhere to an integrity constraint, then this row is said to be in violation of the constraint and is known as an *exception* to the constraint. If any exceptions exist, then the constraint *cannot* be enabled. The rows that violate the constraint must be either updated or deleted in order for the constraint to be enabled.

Exceptions for a specific integrity constraint can be identified while attempting to enable the constraint.

**See Also:** This procedure is discussed in the section "[Exception Reporting](#)".

## On Definition

When you define an integrity constraint in a `CREATE TABLE` or `ALTER TABLE` statement, you can enable the constraint by including the `ENABLE` clause in its definition or disable it by including the `DISABLE` clause in its definition. If neither the `ENABLE` nor the `DISABLE` clause is included in a constraint's definition, Oracle automatically enables the constraint.

### Enabling Constraints

The following `CREATE TABLE` and `ALTER TABLE` statements both define and enable integrity constraints:

```
CREATE TABLE Emp_tab (  
    Empno NUMBER(5) PRIMARY KEY);  
ALTER TABLE Emp_tab  
    ADD PRIMARY KEY (Empno);
```

An `ALTER TABLE` statement that defines and attempts to enable an integrity constraint may fail because rows of the table may violate the integrity constraint. In this case, the statement is rolled back and the constraint definition is not stored and not enabled.

**See Also:** "[Exception Reporting](#)" on page 4-22 for more information about rows that violate integrity constraints.

### Creating Disabled Constraints

The following `CREATE TABLE` and `ALTER TABLE` statements both define and disable integrity constraints:

```
CREATE TABLE Emp_tab (  
    Empno NUMBER(5) PRIMARY KEY DISABLE);  
  
ALTER TABLE Emp_tab  
    ADD PRIMARY KEY (Empno) DISABLE;
```

An `ALTER TABLE` statement that defines and disables an integrity constraints never fails. The definition of the constraint is always allowed because its rule is not enforced.

## Enabling and Disabling Existing Integrity Constraints

Use the `ALTER TABLE` command to

- Enable a disabled constraint, using the `ENABLE` clause
- Disable an enabled constraint, using the `DISABLE` clause

### Enabling Disabled Constraints

The following statements are examples of statements that enable disabled integrity constraints:

```
ALTER TABLE Dept_tab  
    ENABLE CONSTRAINT Dname_ukey;  
  
ALTER TABLE Dept_tab  
    ENABLE PRIMARY KEY  
    ENABLE UNIQUE (Dname)  
    ENABLE UNIQUE (Loc);
```

An `ALTER TABLE` statement that attempts to enable an integrity constraint fails when the rows of the table violate the integrity constraint. In this case, the statement is rolled back and the constraint is not enabled.

**See Also:** 1 "[Exception Reporting](#)" on page 4-22 for more information about rows that violate integrity constraints.

### Disabling Enabled Constraints

The following statements are examples of statements that disable enabled integrity constraints:

```
ALTER TABLE Dept_tab  
    DISABLE CONSTRAINT Dname_ukey;
```

```
ALTER TABLE Dept_tab
  DISABLE PRIMARY KEY
  DISABLE UNIQUE (Dname)
  DISABLE UNIQUE (Loc);
```

---

---

**Tip — Using the Data Dictionary for Reference:** The example statements in the previous sections require that you have some information about a constraint to enable or disable it.

For example, the first statement of each section requires that you know the constraint's name, while the second statement of each section requires that you know the unique key's column list. If you do not have such information, then you can query one of the data dictionary views defined for constraints, `USER_CONSTRAINTS` or `USER_CONS_COLUMNS`; for more information about these views, see "[Listing Integrity Constraint Definitions](#)" on page 4-27 and *Oracle8i Reference*.

---

---

## Enabling and Disabling Key Integrity Constraints

When enabling or disabling `UNIQUE`, `PRIMARY KEY`, and `FOREIGN KEY` integrity constraints, you should be aware of several important issues and prerequisites. `UNIQUE` key and `PRIMARY KEY` constraints are usually managed by the database administrator.

**See Also:** "[Managing FOREIGN KEY Integrity Constraints](#)" on page 4-25 and the *Oracle8i Administrator's Guide*

## Exception Reporting

When you issue a `CREATE TABLE... ENABLE...` or `ALTER TABLE... ENABLE...` statement, and the statement is not successfully executed because integrity constraint exceptions exist, the statement is rolled back and you cannot enable the constraint until all exceptions to the constraint are either updated or deleted. To determine which rows violate the integrity constraint, include the `EXCEPTIONS` option in the `ENABLE` clause of a `CREATE TABLE` or `ALTER TABLE` statement.

**See Also:** *Oracle8i Administrator's Guide* for more information about fixing constraint exceptions.

## Altering Integrity Constraints

In Oracle 8.0, only certain constraint states could be changed using the `ENABLE` or `DISABLE` clauses. With Oracle 8.1, there are expanded capabilities to alter the state of an existing constraint with the `MODIFY CONSTRAINT` clause.

**See Also:** For information on the parameters you can modify, see the `ALTER TABLE` section in *Oracle8i SQL Reference*.

## Examples of MODIFY CONSTRAINT

### Modify Constraint Example #1

```
CREATE TABLE X1_tab (a1 NUMBER CONSTRAINT y CHECK (a1>3) DEFERRABLE DISABLE);

ALTER TABLE X1_tab MODIFY CONSTRAINT Y_cnstrt ENABLE;
ALTER TABLE X1_tab MODIFY CONSTRAINT Y_cnstrt RELY;
ALTER TABLE X1_tab MODIFY CONSTRAINT Y_cnstrt INITIALLY DEFERRED;
ALTER TABLE X1_tab MODIFY CONSTRAINT Y_cnstrt ENABLE NOVALIDATE;
```

### Modify Constraint Example #2

```
CREATE TABLE X1_tab (A1 NUMBER CONSTRAINT Y_cnstrt
NOT NULL DEFERRABLE INITIALLY DEFERRED NORELY DISABLE);

ALTER TABLE X1_tab ADD CONSTRAINT One_cnstrt UNIQUE(A1)
DEFERRABLE INITIALLY IMMEDIATE RELY USING INDEX PCTFREE = 30
ENABLE VALIDATE;

ALTER TABLE X1_tab MODIFY UNIQUE(A1)
INITIALLY DEFERRED NORELY USING INDEX PCTFREE = 40
ENABLE NOVALIDATE;

ALTER TABLE X1_tab MODIFY CONSTRAINT One_cnstrt
INITIALLY IMMEDIATE RELY;
```

### Modify Constraint Example #3

```
CREATE TABLE T1_tab (A1 INT, B1 INT);
```

```
ALTER TABLE T1_tab add CONSTRAINT P1_cnstrt PRIMARY KEY(a1) DISABLE;  
ALTER TABLE T1_tab MODIFY PRIMARY KEY INITIALLY IMMEDIATE  
USING INDEX PCTFREE = 30 ENABLE NOVALIDATE;  
ALTER TABLE T1_tab MODIFY PRIMARY KEY  
USING INDEX PCTFREE = 35 ENABLE;  
ALTER TABLE T1_tab MODIFY PRIMARY KEY ENABLE NOVALIDATE;
```

---

---

**Note:** RELY and NORELY are new states that can be set or reset when a constraint is created or modified.

---

---

## Dropping Integrity Constraints

Drop an integrity constraint if the rule that it enforces is no longer true or if the constraint is no longer needed. Drop an integrity constraint using the ALTER TABLE command and the DROP clause. For example, the following statements drop integrity constraints:

```
ALTER TABLE Dept_tab  
  DROP UNIQUE (Dname);  
ALTER TABLE Dept_tab  
  DROP UNIQUE (Loc);  
  
ALTER TABLE Emp_tab  
  DROP PRIMARY KEY,  
  DROP CONSTRAINT Dept_fkey;  
  
DROP TABLE Emp_tab CASCADE CONSTRAINTS;
```

When dropping UNIQUE, PRIMARY KEY, and FOREIGN KEY integrity constraints, you should be aware of several important issues and prerequisites. UNIQUE and PRIMARY KEY constraints are usually managed by the database administrator.

**See Also:** ["Managing FOREIGN KEY Integrity Constraints"](#) on page 4-25 and the *Oracle8i Administrator's Guide*.

## Managing FOREIGN KEY Integrity Constraints

General information about defining, enabling, disabling, and dropping all types of integrity constraints is given in the previous sections. The following section supplements this information, focusing specifically on issues regarding FOREIGN KEY integrity constraints.

### Defining FOREIGN KEY Integrity Constraints

The following topics are of interest when defining FOREIGN KEY integrity constraints.

#### Matching of Datatypes

When defining referential integrity constraints, the corresponding column names of the dependent and referenced tables do not need to match. However, they must be of the same datatype.

#### Composite Foreign Keys

Because foreign keys reference primary and unique keys of the parent table, and PRIMARY KEY and UNIQUE key constraints are enforced using indexes, composite foreign keys are limited to 32 columns.

#### Implied Referencing of a Primary Key

If the column list is not included in the REFERENCES option when defining a FOREIGN KEY constraint (single column or composite), then Oracle assumes that you intend to reference the primary key of the specified table. Alternatively, you can explicitly specify the column(s) to reference in the parent table within parentheses. Oracle automatically checks to verify that this column list references a primary or unique key of the parent table. If it does not, then an informative error is returned.

#### Privileges Required for FOREIGN KEY Integrity Constraints

To create a FOREIGN KEY constraint, the creator of the constraint must have privileged access to both the parent and the child table.

- **The Parent Table** The creator of the referential integrity constraint must own the parent table or have REFERENCES object privileges on the columns that constitute the parent key of the parent table.
- **The Child Table** The creator of the referential integrity constraint must have the ability to create tables (that is, the CREATE TABLE or CREATE ANY TABLE

system privilege) or the ability to alter the child table (that is, the ALTER object privilege for the child table or the ALTER ANY TABLE system privilege).

In both cases, necessary privileges *cannot* be obtained via a role; they must be explicitly granted to the creator of the constraint.

These restrictions allow:

- The owner of the child table to explicitly decide what constraints are enforced on her or his tables and the other users that can create constraints on her or his tables
- The owner of the parent table to explicitly decide if foreign keys can depend on the primary and unique keys in her tables

### Specifying Referential Actions for Foreign Keys

Oracle allows different types of referential integrity actions to be enforced, as specified with the definition of a FOREIGN KEY constraint:

- **The UPDATE/DELETE No Action Restriction** This action prevents the update or deletion of a parent key if there is a row in the child table that references the key. By default, all FOREIGN KEY constraints enforce the no action restriction; no option needs to be specified when defining the constraint to enforce the no action restriction. For example:

```
CREATE TABLE Emp_tab (  
    FOREIGN KEY (Deptno) REFERENCES Dept_tab);
```

- **The ON DELETE CASCADE Action** This action allows data that references the parent key to be deleted (but not updated). If referenced data in the parent key is deleted, all rows in the child table that depend on the deleted parent key values are also deleted. To specify this referential action, include the ON DELETE CASCADE option in the definition of the FOREIGN KEY constraint. For example:

```
CREATE TABLE Emp_tab (  
    FOREIGN KEY (Deptno) REFERENCES Dept_tab  
    ON DELETE CASCADE);
```

- **The ON DELETE SET NULL Action** This action allows data that references the parent key to be deleted (but not updated). If referenced data in the parent key is deleted, all rows in the child table that depend on the deleted parent key values have their foreign keys set to null. To specify this referential action, include the ON DELETE SET NULL option in the definition of the FOREIGN KEY constraint. For example:

```
CREATE TABLE Emp_tab (
    FOREIGN KEY (Deptno) REFERENCES Dept_tab
    ON DELETE SET NULL);
```

## Enabling FOREIGN KEY Integrity Constraints

FOREIGN KEY integrity constraints cannot be enabled if the referenced primary or unique key's constraint is not present or not enabled.

## Listing Integrity Constraint Definitions

The data dictionary contains the following views that relate to integrity constraints:

- ALL\_CONSTRAINTS
- ALL\_CONS\_COLUMNS
- CONSTRAINT\_COLUMNS
- CONSTRAINT\_DEFS
- USER\_CONSTRAINTS
- USER\_CONS\_COLUMNS
- USER\_CROSS\_REFS
- DBA\_CONSTRAINTS
- DBA\_CONS\_COLUMNS
- DBA\_CROSS\_REFS

**See Also:** Refer to *Oracle8i Reference* for detailed information about each view.

## Examples

Consider the following CREATE TABLE statements that define a number of integrity constraints:

```
CREATE TABLE Dept_tab (
    Deptno    NUMBER(3) PRIMARY KEY,
    Dname     VARCHAR2(15),
    Loc       VARCHAR2(15),
    CONSTRAINT Dname_ukey UNIQUE (Dname, Loc),
    CONSTRAINT LOC_CHECK1
    CHECK (Loc IN ('NEW YORK', 'BOSTON', 'CHICAGO')));
```

```
CREATE TABLE Emp_tab (
    Empno    NUMBER(5) PRIMARY KEY,
    Ename    VARCHAR2(15) NOT NULL,
    Job      VARCHAR2(10),
    Mgr      NUMBER(5) CONSTRAINT Mgr_fkey
           REFERENCES Emp_tab ON DELETE CASCADE,
    Hiredate DATE,
    Sal      NUMBER(7,2),
    Comm     NUMBER(5,2),
    Deptno   NUMBER(3) NOT NULL
    CONSTRAINT Dept_fkey REFERENCES Dept_tab);
```

**Example 1: Listing All of Your Accessible Constraints** The following query lists all constraints defined on all tables accessible to the user:

```
SELECT Constraint_name, Constraint_type, Table_name,
       R_constraint_name
FROM User_constraints;
```

Considering the example statements at the beginning of this section, a list similar to the one below is returned:

CONSTRAINT_NAME	C	TABLE_NAME	R_CONSTRAINT_NAME
SYS_C00275	P	DEPT_TAB	
DNAME_UKEY	U	DEPT_TAB	
LOC_CHECK1	C	DEPT_TAB	
SYS_C00278	C	EMP_TAB	
SYS_C00279	C	EMP_TAB	
SYS_C00280	P	EMP_TAB	
MGR_FKEY	R	EMP_TAB	SYS_C00280
DEPT_FKEY	R	EMP_TAB	SYS_C00275

Notice the following:

- Some constraint names are user specified (such as DNAME\_UKEY), while others are system specified (such as SYS\_C00275).
- Each constraint type is denoted with a different character in the CONSTRAINT\_TYPE column. The table below summarizes the characters used for each constraint type.

<b>Constraint Type</b>	<b>Character</b>
PRIMARY KEY	P
UNIQUE KEY	U
FOREIGN KEY	R
CHECK, NOT NULL	C

---

**Note:** An additional constraint type is indicated by the character "V" in the CONSTRAINT\_TYPE column. This constraint type corresponds to constraints created by the WITH CHECK OPTION for views. See [Chapter 2, "Managing Schema Objects"](#) for more information about views and the WITH CHECK OPTION.

---

**Example 2: Distinguishing NOT NULL Constraints from CHECK Constraints** In the previous example, several constraints are listed with a constraint type of "C". To distinguish which constraints are NOT NULL constraints and which are CHECK constraints in the EMP\_TAB and DEPT\_TAB tables, issue the following query:

```
SELECT Constraint_name, Search_condition
       FROM User_constraints
       WHERE (Table_name = 'DEPT_TAB' OR Table_name = 'EMP_TAB') AND
              Constraint_type = 'C';
```

Considering the example CREATE TABLE statements at the beginning of this section, a list similar to the one below is returned:

```
CONSTRAINT_NAME  SEARCH_CONDITION
-----
LOC_CHECK1       loc IN ('NEW YORK', 'BOSTON', 'CHICAGO')
SYS_C00278       ENAME IS NOT NULL
SYS_C00279       DEPINO IS NOT NULL
```

Notice the following:

- NOT NULL constraints are clearly identified in the SEARCH\_CONDITION column.
- The conditions for user-defined CHECK constraints are explicitly listed in the SEARCH\_CONDITION column.

**Example 3: Listing Column Names that Constitute an Integrity Constraint** The following query lists all columns that constitute the constraints defined on all tables accessible to you, the user:

```
SELECT Constraint_name, Table_name, Column_name
       FROM User_cons_columns;
```

Considering the example statements at the beginning of this section, a list similar to the one below is returned:

CONSTRAINT_NAME	TABLE_NAME	COLUMN_NAME
-----	-----	-----
DEPT_FKEY	EMP_TAB	DEPTNO
DNAME_UKEY	DEPT_TAB	DNAME
DNAME_UKEY	DEPT_TAB	LOC
LOC_CHECK1	DEPT_TAB	LOC
MGR_FKEY	EMP_TAB	MGR
SYS_C00275	DEPT_TAB	DEPTNO
SYS_C00278	EMP_TAB	ENAME
SYS_C00279	EMP_TAB	DEPTNO
SYS_C00280	EMP_TAB	EMPNO

---

## Selecting an Index Strategy

This chapter discusses the procedures necessary to create and manage the different types of objects contained in a user's schema. The topics include:

- [Managing Indexes](#)
- [Function-Based Indexes](#)
- [Managing Clusters, Clustered Tables, and Cluster Indexes](#)
- [Managing Hash Clusters and Clustered Tables](#)

**See Also:** Specific information is described in the following locations:

- Procedures, functions, and packages — [Chapter 9](#)
- Object types — *Oracle8i Application Developer's Guide - Object-Relational Features*
- Dependency information — [Chapter 9](#)
- If you use symmetric replication, then see *Oracle8i Replication* for information on managing schema objects, such as snapshots.

### Managing Indexes

Indexes are used in Oracle to provide quick access to rows in a table. Indexes provide faster access to data for operations that return a small portion of a table's rows.

Oracle does not limit the number of indexes you can create on a table. However, you should consider the performance benefits of indexes and the needs of your database applications to determine which columns to index.

The following sections explain how to create, alter, and drop indexes using SQL commands. Some simple guidelines to follow when managing indexes are included.

**See Also:** See *Oracle8i Designing and Tuning for Performance* for performance implications of index creation.

### **Create Indexes After Inserting Table Data**

With one notable exception, you should usually create indexes after you have inserted or loaded (using SQL\*Loader or Import) data into a table. It is more efficient to insert rows of data into a table that has no indexes and create the indexes later. If you create indexes before table data is loaded, then every index must be updated every time you insert a row into the table. The exception to this rule is that you must create an index for a cluster before you insert any data into the cluster.

When you create an index on a table that already has data, Oracle must use sort space to create the index. Oracle uses the sort space in memory allocated for the creator of the index (the amount per user is determined by the initialization parameter `SORT_AREA_SIZE`), but must also swap sort information to and from temporary segments allocated on behalf of the index creation. If the index is extremely large, it might be beneficial to complete the following steps:

1. Create a new temporary tablespace using the `CREATE TABLESPACE` command.
2. Use the `TEMPORARY TABLESPACE` option of the `ALTER USER` command to make this your new temporary tablespace.
3. Create the index using the `CREATE INDEX` command.
4. Drop this tablespace using the `DROP TABLESPACE` command. Then use the `ALTER USER` command to reset your temporary tablespace to your original temporary tablespace.

Under certain conditions, you can load data into a table with the SQL\*Loader "direct path load", and an index can be created as data is loaded.

**See Also:** *Oracle8i Utilities*

**Index the Correct Tables and Columns** Use the following guidelines for determining when to create an index:

- Create an index if you frequently want to retrieve less than 15% of the rows in a large table. The percentage varies greatly according to the relative speed of a table scan and how clustered the row data is about the index key. The faster the table scan, the lower the percentage; the more clustered the row data, the higher the percentage.
- Index columns used for joins to improve performance on joins of multiple tables.

**See Also:** Primary and unique keys automatically have indexes, but you might want to create an index on a foreign key; see [Chapter 4, "Maintaining Data Integrity"](#) for more information.

- Small tables do not require indexes; if a query is taking too long, then the table might have grown from small to large.

Some columns are strong candidates for indexing. Columns with one or more of the following characteristics are candidates for indexing:

- Values are relatively unique in the column.
- There is a wide range of values.
- The column contains many nulls, but queries often select all rows having a value. In this case, the following phrase:

```
WHERE COL_X > -9.99 *power(10,125)
```

is preferable to

```
WHERE COL_X IS NOT NULL
```

This is because the first uses an index on COL\_X (assuming that COL\_X is a numeric column).

Columns with the following characteristics are less suitable for indexing:

- The column has few distinct values (for example, a column for the sex of employees).
- There are many nulls in the column and you do not search on the non-null values.

LONG and LONG RAW columns cannot be indexed.

The size of a single index entry cannot exceed roughly one-half (minus some overhead) of the available space in the data block. Consult with the database administrator for assistance in determining the space required by an index.

**Limit the Number of Indexes per Table** A table can have any number of indexes. However, the more indexes, the more overhead is incurred as the table is altered. When rows are inserted or deleted, all indexes on the table must be updated. When a column is updated, all indexes on the column must be updated.

Thus, there is a trade-off between speed of retrieval for queries on a table and speed of accomplishing updates on the table. For example, if a table is primarily read-only, then more indexes might be useful; but, if a table is heavily updated, then fewer indexes might be preferable.

**Order Index Columns for Performance** The order in which columns are named in the CREATE INDEX command does not need to correspond to the order in which they appear in the table. However, the order of columns in the CREATE INDEX statement is significant because query performance can be affected by the order chosen. In general, you should put the column expected to be used most often first in the index.

For example, assume the columns of the VENDOR\_PARTS table are as shown in [Figure 5-1](#).

**Figure 5-1 The VENDOR\_PARTS Table**

Table VENDOR_PARTS		
VEND ID	PART NO	UNIT COST
1012	10-440	.25
1012	10-441	.39
1012	457	4.95
1010	10-440	.27
1010	457	5.10
1220	08-300	1.33
1012	08-300	1.19
1292	457	5.28

Assume that there are five vendors, and each vendor has about 1000 parts.

Suppose that the VENDOR\_PARTS table is commonly queried by SQL statements such as the following:

```
SELECT * FROM vendor_parts
    WHERE part_no = 457 AND vendor_id = 1012;
```

To increase the performance of such queries, you might create a composite index putting the most selective column first; that is, the column with the *most* values:

```
CREATE INDEX ind_vendor_id
  ON vendor_parts (part_no, vendor_id);
```

Indexes speed retrieval on any query using the *leading portion* of the index. So in the above example, queries with `WHERE` clauses using only the `PART_NO` column also note a performance gain. Because there are only five distinct values, placing a separate index on `VENDOR_ID` would serve no purpose.

## Creating Indexes

You can create an index for a table to improve the performance of queries issued against the corresponding table. You can also create an index for a cluster. You can create a *composite* index on multiple columns up to a maximum of 32 columns. A composite index key cannot exceed roughly one-half (minus some overhead) of the available space in the data block.

Oracle automatically creates an index to enforce a `UNIQUE` or `PRIMARY KEY` integrity constraint. In general, it is better to create such constraints to enforce uniqueness and not explicitly use the obsolete `CREATE UNIQUE INDEX` syntax.

Use the SQL command `CREATE INDEX` to create an index. The following statement

```
CREATE INDEX emp_ename ON Emp_tab(ename)
  TABLESPACE users
  STORAGE (INITIAL 20K
           NEXT 20k
           PCTINCREASE 75)
  PCTFREE 0;
```

Notice that several storage settings are explicitly specified for the index.

### Privileges Required to Create an Index

To create a new index, you must own, or have the `INDEX` object privilege for, the corresponding table. The schema that contains the index must also have a quota for the tablespace intended to contain the index, or the `UNLIMITED TABLESPACE` system privilege. To create an index in another user's schema, you must have the `CREATE ANY INDEX` system privilege.

## Dropping Indexes

You might drop an index for the following reasons:

- The index is not providing anticipated performance improvements for queries issued against the associated table (the table is very small, or there are many rows in the table but very few index entries, etc.).
- Applications do not contain queries that use the index.
- The index is no longer needed and must be dropped before being rebuilt.

When you drop an index, all extents of the index's segment are returned to the containing tablespace and become available for other objects in the tablespace.

Use the SQL command `DROP INDEX` to drop an index. For example, to drop the `EMP_ENAME` index, enter the following statement:

```
DROP INDEX Emp_ename;
```

If you drop a table, then all associated indexes are dropped.

**Privileges Required to Drop an Index** To drop an index, the index must be contained in your schema or you must have the `DROP ANY INDEX` system privilege.

## Function-Based Indexes

A function-based index is an index built on an expression. It extends your indexing capabilities beyond indexing on a column. A function-based index increases the variety of ways in which you can access data.

---

---

**Note:** You can create function-based indexes only if you are using the Oracle8i release, or higher.

---

---

The expression used in a function-based index can be an arithmetic expression or an expression that contains a PL/SQL function, package function, C callout, or SQL function. Function-based indexes also support linguistic sorts based on linguistic sort keys (collation), efficient linguistic collation of SQL statements, and case-insensitive sorts.

Like other indexes, function-based indexes improve query performance. For example, if you need to access a computationally complex expression often, then you can store it in an index. Then when you need to access the expression, it is already computed. You can find a detailed description of the advantages of function-based indexes in ["Using Function-Based Indexes"](#) on page 5-7.

Function-based indexes have all of the same properties as indexes on columns. However, unlike indexes on columns which can be used by both cost-based and rule-based optimization, function-based indexes can be used only by cost-based optimization. Other restrictions on function-based indexes are described in "[Requirements and Restrictions for Function-Based Indexes](#)" on page 5-12.

**See Also:** For more information on function-based indexes, see *Oracle8i Concepts*. For information on creating function-based indexes, see *Oracle8i Administrator's Guide*.

## Using Function-Based Indexes

The following list describes the advantages of function-based indexes in greater detail:

- **Increase the number of situations where the optimizer can perform a range scan instead of a full table scan.** For example: consider the expression in the WHERE clause below:

```
CREATE INDEX idx ON Example_tab(Column_a + Column_b);  
SELECT * FROM Example_tab WHERE Column_a + Column_b < 10;
```

In the CREATE INDEX statement, `idx` is the name of the index, `Example_tab` is the name of the table, and `column_a` and `column_b` represent columns. The optimizer can use a range scan for this query because the index is built on `(column_a + column_b)`. Range scans typically produce fast response times if the predicate has low selectivity (that is, if the predicate selects less than 15% of the rows of a large table). In addition, the optimizer can estimate selectivity of predicates involving expressions more accurately if the expressions are materialized in a function-based index (expressions of function-based indexes are represented as virtual columns and ANALYZE can build histograms on such columns).

- **Precompute the value of a computationally intensive function and store it in the index.** If you have a computationally intensive expression that you access often, then you can store it in an index. When you need to access it, the value is already computed. This can greatly improve query execution performance.
- **Create indexes on object columns and REF columns.** Methods that describe objects can be used as functions on which to build indexes. For example, you can use the MAP method to build indexes on an object type column.

- **Create more powerful sorts.** You can perform case-insensitive sorts with the `UPPER` and `LOWER` functions, descending order sorts with the `DESC` keyword, and linguistic-based sorts with the `NLSSORT` function.

---

---

**Note:** The `DESC` keyword in the `CREATE INDEX` statement is no longer ignored. Oracle sorts columns with the `DESC` keyword in descending order. Such indexes are treated as function-based indexes. Descending indexes cannot be bitmapped or reverse, and cannot be used in bitmapped optimizations. To get the pre-Oracle 8.1 release `DESC` functionality, remove the `DESC` keyword from the `CREATE INDEX` statement.

---

---

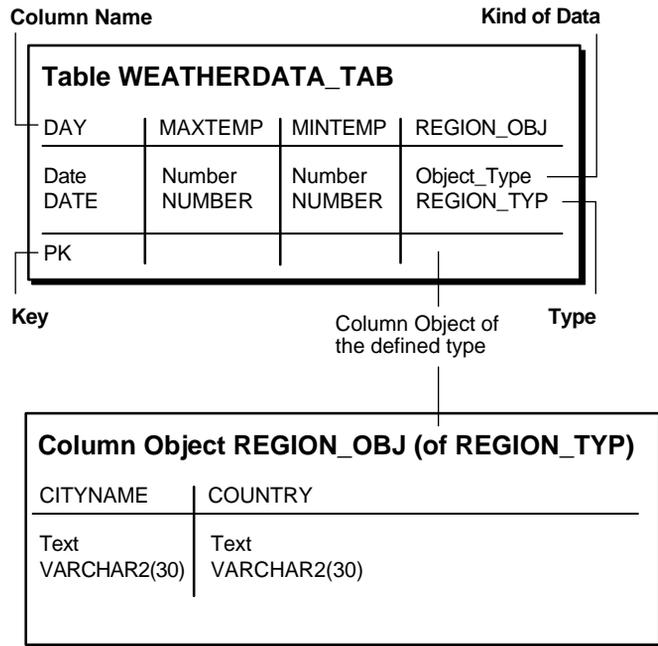
**See Also:** For examples of how to use function-based indexes, see the *Oracle8i Administrator's Guide*.

### Example

As an example, consider a weather research institute that maintains tables of weather data for various cities. Some of their projects include tracking daily temperature fluctuations throughout the year. Other projects include tracking fluctuations as a function of the city's distance from the equator. By building indexes on the complex functions that they want to calculate, the institute can optimize the execution of the queries they submit. The following section contains examples of indexes that could be created and the queries that could use them.

The table, `Weatherdata_tab`, contains columns for the minimum daily temperature (`Mintemp`), maximum daily temperature (`Maxtemp`), the day the temperature was recorded (`Day`), and the Region (`Region_Obj`). `Region_Obj` is an object column that contains columns for country (`Country`) and city (`Cityname`). [Figure 5-2](#) illustrates the `Weatherdata_tab` schema.

Figure 5–2 WEATHERDATA\_TAB Schema Design



An index is created that calculates the difference in temperature for the cities in the tables. A query that could use the `delta_index` index returns the contents of the table for temperature differences less than 20:

---

**Note:** You may need to set up data structures similar to the following for certain examples to work:

```
CREATE OR REPLACE FUNCTION distance_from_equator(input
NUMBER) RETURN NUMBER DETERMINISTIC IS
    distance NUMBER;
BEGIN
    distance := 100000;
    RETURN (distance);
END;
```

---

```
CREATE INDEX Delta_index
ON Weatherdata_tab (Maxtemp - Mintemp);
```

```
SELECT *
FROM Weatherdata_tab
WHERE (Maxtemp - Mintemp) < '20';
```

An index is created that calls the object method `distance_from_equator` to calculate the distance from the equator for each city in the table. The method is applied to the object column `Region_Obj`. A query that could use the `distance_index` index returns the names of the cities that are at a distance greater than 1000 miles from the equator:

```
CREATE INDEX Distance_index
ON Weatherdata_tab (Distance_from_equator (Reg_obj));
```

```
SELECT *
FROM Weatherdata_tab
WHERE (Distance_from_equator (Reg_Obj)) > '1000';
```

An index is created that satisfies the queries of German-speaking users that sorts temperature data by city name. A query that could use the `City_index` index returns the contents of the table, ordered by city name. The German sort order for city name is used. Note that in the `SELECT` statement, a `WHERE` clause is not needed. This is because in a German session, `NLS_SORT` is set to `German` and `NLS_COMP` is set to `ANSI`.

```
CREATE INDEX City_index
ON Weatherdata_tab (NLSSORT(Cityname, 'NLS_SORT=German'));
```

```
SELECT *
FROM Weatherdata_tab WHERE Cityname IS NOT NULL
ORDER BY Cityname;
```

An index is created on the difference between the maximum and minimum temperatures, and on the maximum temperature. The result of the difference is sorted in descending order. A query that could use the `compare_index` index returns the contents of the table that satisfy the condition where the difference is less than 20 and the maximum temperature is greater than 75.

```
CREATE INDEX compare_index
ON Weatherdata_tab ((Maxtemp - Mintemp) DESC, Maxtemp);
```

```
SELECT *
FROM Weatherdata_tab WHERE ((Maxtemp - Mintemp) < '20' AND Maxtemp > '75');
```

## Example Function-Based Indexes

### Example 1:

The following command creates a function-based index `IDX` on table `EMP_TAB`, for efficient case-insensitive searches.

```
CREATE INDEX Idx ON Emp_tab (UPPER(Ename));
```

The `SELECT` command uses the function-based index on `UPPER(e_name)` to return all of the employees with name like `:KEYCOL`.

```
SELECT *
FROM Emp_tab
WHERE UPPER(Ename) like :KEYCOL;
```

### Example 2:

The following command creates a function-based index `IDX` on table `Fbi_tab` where `A`, `B`, and `C` represent columns.

```
CREATE INDEX Idx
On Fbi_tab (A + B * (C - 1), A, B);
```

The `SELECT` statement can either use index range scan (notice that the expression is a prefix of index `IDX`) or index fast full scan (which may be preferable if the index has specified a high parallel degree).

```
SELECT a
FROM Fbi_tab
Where A + B * (C - 1) < 100;
```

### Example 3:

This example demonstrates how a function-based index can be used to support an NLS Sort Index. Given a string, the `NLSSORT` function returns a sort key. The following `CREATE INDEX` statement creates an `NLS_SORT` sort on table `NLS_TAB` with collation sequence `GERMAN`.

```
CREATE INDEX Nls_index
ON Nls_tab (NLSSORT(Name, 'NLS_SORT = German'));
```

The `SELECT` statement selects all of the contents of the table and orders it by `NAME`. The rows are ordered using the German collation sequence.

```
SELECT *
```

```
FROM Nls_tab WHERE Name IS NOT NULL  
ORDER BY Name;
```

## Requirements and Restrictions for Function-Based Indexes

Note the following requirements and restrictions for function-based indexes:

- Only cost-based optimization can use function-based indexes.
- A PL/SQL function, either a top level function or a package-level function, used in the index expression must be declared as `DETERMINISTIC`. There is no error checking whether or not a subprogram is qualified as `DETERMINISTIC`. You must ensure that the subprogram is `DETERMINISTIC`.

The following semantic rules demonstrate how to use the keyword `DETERMINISTIC`:

- A top level subprogram can be declared as `DETERMINISTIC`.
- A `PACKAGE` level subprogram can be declared as `DETERMINISTIC` in the `PACKAGE` specification but not in the `PACKAGE BODY`. Errors are raised if `DETERMINISTIC` is used inside a `PACKAGE BODY`.
- A private subprogram (declared inside another subprogram or a `PACKAGE BODY`) cannot be declared as `DETERMINISTIC`.
- A `DETERMINISTIC` subprogram can call another subprogram whether the called program is declared as `DETERMINISTIC` or not.
- Function-based indexes cannot be built on `LOB` columns, nested tables, or varrays.
- Expressions used in a function-based index should reference only columns in a row in the table. Hence, these expressions cannot contain any aggregate functions.
- You must have the initialization parameters `COMPATIBLE` set to 8.1.0.0.0 or higher, `QUERY_REWRITE_ENABLED=TRUE`, and `QUERY_REWRITE_INTEGRITY=TRUSTED`.
- You must analyze the table or index before the index is used.
- Bitmap optimizations cannot be used on descending indexes.
- Function-based indexes are not used when `OR-expansion` is done.
- The index function cannot be marked `NOT NULL`. To avoid a full table scan, you must ensure that the query cannot fetch null values.

- Function-based indexes that return `VARCHAR2` or `RAW` data types from a PL/SQL function are not permitted due to length restrictions. A possible work around is to use substrings to limit the size of the function's output. For example:

---

---

**Note:** You may need to set up data structures similar to the following for certain examples to work:

```
CREATE OR REPLACE FUNCTION x(input IN VARCHAR2)
RETURN VARCHAR2 AS
output VARCHAR2(12);
BEGIN
    output :=input;
    RETURN (output);
END;
```

```
SELECT SUBSTR(x('hello'),1,100) FROM DUAL;
```

---

---

```
SUBSTR (F(X), 1, 100)
```

Where `F(X)` represents the PL/SQL function. The `SUBSTR` command would need to be used for the function when creating the index and when referencing the function in queries.

## Managing Clusters, Clustered Tables, and Cluster Indexes

Because clusters store related rows of different tables together in the same data blocks, two primary benefits are achieved when clusters are properly used:

- Disk I/O is reduced and access time improves for joins of clustered tables.
- In a cluster, a *cluster key value* (the related value) is only stored once, no matter how many rows of different tables contain the value. Therefore, less storage may be required to store related table data in a cluster than is necessary in non-clustered table format.

### Guidelines for Creating Clusters

Some guidelines for creating clusters are outlined below.

**See Also:** For performance characteristics, see *Oracle8i Designing and Tuning for Performance*.

**Choose Appropriate Tables to Cluster** Use clusters to store one or more tables that are primarily queried (not predominantly inserted into or updated), and for which queries often join data of multiple tables in the cluster or retrieve related data from a single table.

**Choose Appropriate Columns for the Cluster Key** Choose cluster key columns carefully. If multiple columns are used in queries that join the tables, then make the cluster key a composite key. In general, the same column characteristics that make a good index apply for cluster indexes.

**See Also:** "[Index the Correct Tables and Columns](#)" on page 5-2 has more information about these guidelines.

A good cluster key has enough unique values so that the group of rows corresponding to each key value fills approximately one data block. Too few rows per cluster key value can waste space and result in negligible performance gains. Cluster keys that are so specific that only a few rows share a common value can cause wasted space in blocks, unless a small `SIZE` was specified at cluster creation time.

Too many rows per cluster key value can cause extra searching to find rows for that key. Cluster keys on values that are too general (for example, `MALE` and `FEMALE`) result in excessive searching and can result in worse performance than with no clustering.

A cluster index cannot be unique or include a column defined as `LONG`.

## Performance Considerations

Also note that clusters can reduce the performance of DML statements (`INSERTS`, `UPDATES`, and `DELETES`) as compared to storing a table separately with its own index. These disadvantages relate to the use of space and the number of blocks that must be visited to scan a table. Because multiple tables share each block, more blocks must be used to store a clustered table than if that same table were stored non-clustered. You should decide about using clusters with these trade-offs in mind.

To identify data that would be better stored in clustered form than in non-clustered form, look for tables that are related via referential integrity constraints, and tables that are frequently accessed together using `SELECT` statements that join data from two or more tables. If you cluster tables on the columns used to join table data, then you reduce the number of data blocks that must be accessed to process the query; all the rows needed for a join on a cluster key are in the same block. Therefore, query performance for joins is improved.

Similarly, it may be useful to cluster an individual table. For example, the `EMP_TAB` table could be clustered on the `DEPTNO` column to cluster the rows for employees in the same department. This would be advantageous if applications commonly process rows, department by department.

Like indexes, clusters do not affect application design. The existence of a cluster is transparent to users and to applications. Data stored in a clustered table is accessed via SQL just like data stored in a non-clustered table.

## Creating Clusters, Clustered Tables, and Cluster Indexes

Use a cluster to store one or more tables that are frequently joined in queries. Do not use a cluster to cluster tables that are frequently accessed individually.

Once you create a cluster, tables can be created in the cluster. However, before you can insert any rows into the clustered tables, you must create a cluster index. The use of clusters does not affect the creation of additional indexes on the clustered tables; you can create and drop them as usual.

Use the SQL command `CREATE CLUSTER` to create a cluster. The following statement creates a cluster named `EMP_DEPT`, which stores the `EMP_TAB` and `DEPT_TAB` tables, clustered by the `DEPTNO` column:

```
CREATE CLUSTER Emp_dept (Deptno NUMBER(3))
```

```
PCTUSED 80
PCTFREE 5;
```

Create a table in a cluster using the SQL command `CREATE TABLE` with the `CLUSTER` option. For example, the `EMP_TAB` and `DEPT_TAB` tables can be created in the `EMP_DEPT` cluster using the following statements:

```
CREATE TABLE Dept_tab (
  Deptno NUMBER(3) PRIMARY KEY,
  . . . )
CLUSTER Emp_dept (Deptno);

CREATE TABLE Emp_tab (
  Empno NUMBER(5) PRIMARY KEY,
  Ename VARCHAR2(15) NOT NULL,
  . . .
  Deptno NUMBER(3) REFERENCES Dept_tab)
CLUSTER Emp_dept (Deptno);
```

A table created in a cluster is contained in the schema specified in the `CREATE TABLE` statement; a clustered table might not be in the same schema that contains the cluster.

You must create a cluster index before any rows can be inserted into any clustered table. For example, the following statement creates a cluster index for the `EMP_DEPT` cluster:

```
CREATE INDEX Emp_dept_index
ON CLUSTER Emp_dept
INITRANS 2
MAXTRANS 5
PCTFREE 5;
```

---

---

**Note:** A cluster index cannot be unique. Furthermore, Oracle is not guaranteed to enforce uniqueness of columns in the cluster key if they have `UNIQUE` or `PRIMARY KEY` constraints.

---

---

The cluster key establishes the relationship of the tables in the cluster.

### Privileges Required to Create a Cluster, Clustered Table, and Cluster Index

To create a cluster in your schema, you must have the `CREATE CLUSTER` system privilege and a quota for the tablespace intended to contain the cluster or the

UNLIMITED TABLESPACE system privilege. To create a cluster in another user's schema, you must have the CREATE ANY CLUSTER system privilege, and the owner must have a quota for the tablespace intended to contain the cluster or the UNLIMITED TABLESPACE system privilege.

To create a table in a cluster, you must have either the CREATE TABLE or CREATE ANY TABLE system privilege. You do not need a tablespace quota or the UNLIMITED TABLESPACE system privilege to create a table in a cluster.

To create a cluster index, your schema must contain the cluster, and you must have the following privileges:

- The CREATE ANY INDEX system privilege or, if you own the cluster, the CREATE INDEX privilege
- A quota for the tablespace intended to contain the cluster index, or the UNLIMITED TABLESPACE system privilege

## Manually Allocating Storage for a Cluster

Oracle dynamically allocates additional extents for the data segment of a cluster, as required. In some circumstances, you might want to explicitly allocate an additional extent for a cluster. For example, when using the Oracle Parallel Server, an extent of a cluster can be allocated explicitly for a specific instance.

You can allocate a new extent for a cluster using the SQL command ALTER CLUSTER with the ALLOCATE EXTENT option.

**See Also:** *Oracle8i Parallel Server Documentation Set: Oracle8i Parallel Server Concepts; Oracle8i Parallel Server Setup and Configuration Guide; Oracle8i Parallel Server Administration, Deployment, and Performance*

## Dropping Clusters, Clustered Tables, and Cluster Indexes

Drop a cluster if the tables currently within the cluster are no longer necessary. When you drop a cluster, the tables within the cluster and the corresponding cluster index are dropped; all extents belonging to both the cluster's data segment and the index segment of the cluster index are returned to the containing tablespace and become available for other segments within the tablespace.

You can individually drop clustered tables without affecting the table's cluster, other clustered tables, or the cluster index. Drop a clustered table in the same manner as a non-clustered table—use the SQL command DROP TABLE.

See "Dropping Tables" for more information about individually dropping tables.

---

---

**Note:** When you drop a single clustered table from a cluster, each row of the table must be deleted from the cluster. To maximize efficiency, if you intend to drop the entire cluster including all tables, then use the `DROP CLUSTER` command with the `INCLUDING TABLES` option.

You should only use the `DROP TABLE` command to drop an individual table from a cluster when the rest of the cluster is going to remain.

---

---

You can drop a cluster index without affecting the cluster or its clustered tables. However, you cannot use a clustered table if it does not have a cluster index. Cluster indexes are sometimes dropped as part of the procedure to rebuild a fragmented cluster index.

**See Also:** ["Dropping Indexes"](#) on page 5-5

To drop a cluster that contains no tables, as well as its cluster index, if present, use the SQL command `DROP CLUSTER`. For example, the following statement drops the empty cluster named `EMP_DEPT`:

```
DROP CLUSTER Emp_dept;
```

If the cluster contains one or more clustered tables, and if you intend to drop the tables as well, then add the `INCLUDING TABLES` option of the `DROP CLUSTER` command. For example:

```
DROP CLUSTER Emp_dept INCLUDING TABLES;
```

If you do not include the `INCLUDING TABLES` option, and if the cluster contains tables, then an error is returned.

If one or more tables in a cluster contain primary or unique keys that are referenced by `FOREIGN KEY` constraints of tables outside the cluster, then you cannot drop the cluster unless you also drop the dependent `FOREIGN KEY` constraints. Use the `CASCADE CONSTRAINTS` option of the `DROP CLUSTER` command, as in

```
DROP CLUSTER Emp_dept INCLUDING TABLES CASCADE CONSTRAINTS;
```

An error is returned if the above option is not used in the appropriate situation.

### Privileges Required to Drop a Cluster

To drop a cluster, your schema must contain the cluster, or you must have the `DROP ANY CLUSTER` system privilege. You do not have to have any special privileges to drop a cluster that contains tables, even if the clustered tables are not owned by the owner of the cluster.

## Managing Hash Clusters and Clustered Tables

The following sections explain how to create, alter, and drop hash clusters and clustered tables using SQL commands.

### Creating Hash Clusters and Clustered Tables

A hash cluster is used to store individual tables or a group of clustered tables that are static and often queried by equality queries. Once you create a hash cluster, you can create tables. To create a hash cluster, use the SQL command `CREATE CLUSTER`. The following statement creates a cluster named `TRIAL_CLUSTER` that is used to store the `TRIAL_TAB` table, clustered by the `TRIALNO` column:

---



---

**Note:** You may need to use a setup similar to the following for certain examples to work:

```
ALTER TABLESPACE SYSTEM ADD DATAFILE 'disk1:moredata1' SIZE 50K
AUTOEXTEND ON;
```

---



---

```
CREATE CLUSTER Trial_cluster (
  Trialno NUMBER(5,0))
  PCTUSED 80
  PCTFREE 5
  SIZE 2K
  HASH IS Trialno HASHKEYS 100000;

CREATE TABLE Trial_tab (
  Trialno NUMBER(5) PRIMARY KEY,
  ...)
  CLUSTER Trial_cluster (Trialno);
```

## Controlling Space Usage Within a Hash Cluster

When you create a hash cluster, it is important that you correctly choose the cluster key and set the `HASH IS`, `SIZE`, and `HASHKEYS` parameters to achieve the desired performance and space usage for the cluster. The following sections provide guidance, as well as examples of setting these parameters.

### Choosing the Key

Choosing the correct cluster key is dependent on the most common types of queries issued against the clustered tables. For example, consider the `EMP_TAB` table in a hash cluster. If queries often select rows by employee number, then the `EMPNO` column should be the cluster key; if queries often select rows by department number, then the `DEPTNO` column should be the cluster key. For hash clusters that contain a single table, the cluster key is typically the entire primary key of the contained table. A hash cluster with a composite key must use Oracle's internal hash function.

### Setting `HASH IS`

Only specify the `HASH IS` parameter if the cluster key is a single column of the `NUMBER` datatype and contains uniformly distributed integers. If the above conditions apply, then you can distribute rows in the cluster such that each unique cluster key value hashes to a unique hash value (with no collisions). If the above conditions do not apply, you should use the internal hash function.

## Dropping Hash Clusters

Drop a hash cluster using the SQL command `DROP CLUSTER`:

```
DROP CLUSTER Emp_dept;
```

Drop a table in a hash cluster using the SQL command `DROP TABLE`. The implications of dropping hash clusters and tables in hash clusters are the same as for index clusters.

**See Also:** ["Dropping Clusters, Clustered Tables, and Cluster Indexes"](#) on page 5-17

## When to Use Hashing

Storing a table in a hash cluster is an alternative to storing the same table with an index. Hashing is useful in the following situations:

- Most queries are equality queries on the cluster key. For example:

```
SELECT . . . WHERE Cluster_key = . . . ;
```

In such cases, the cluster key in the equality condition is hashed, and the corresponding hash key is usually found with a single read. With an indexed table, the key value must first be found in the index (usually several reads), and then the row is read from the table (another read).

- The table or tables in the hash cluster are primarily static in size such that you can determine the number of rows and amount of space required for the tables in the cluster. If tables in a hash cluster require more space than the initial allocation for the cluster, then performance degradation can be substantial because overflow blocks are required.
- A hash cluster with the `HASH IS col`, `HASHKEYS n`, and `SIZE m` clauses is an ideal representation for an array (table) of  $n$  items (rows) where each item consists of  $m$  bytes of data. For example:

```
ARRAY X[100] OF NUMBER(8)
```

This could be represented as the following:

```
CREATE CLUSTER C(Subscript INTEGER)
    HASH IS Subscript HASHKEYS 100 SIZE 100;

CREATE TABLE X(Subscript NUMBER(2), Value NUMBER(8))
    CLUSTER C(Subscript);
```

Alternatively, hashing is not advantageous in the following situations:

- Most queries on the table retrieve rows over a range of cluster key values. For example, in full table scans, or queries:

```
SELECT . . . WHERE Cluster_key < . . . ;
```

A hash function cannot be used to determine the location of specific hash keys; instead, the equivalent of a full table scan must be done to fetch the rows for the query. With an index, key values are ordered in the index, so cluster key values that satisfy the `WHERE` clause of a query can be found with relatively few I/Os.

- A table is not static, but is continually growing. If a table grows without limit, then the space required over the life of the table (thus, of its cluster) cannot be predetermined.

- Applications frequently perform full table scans on the table and the table is sparsely populated. A full table scan in this situation takes longer under hashing.
- You cannot afford to preallocate the space the hash cluster will eventually need.

In most cases, you should decide (based on the above information) whether to use hashing or indexing. If you use indexing, consider whether it is best to store a table individually or as part of a cluster.

**See Also:** ["Guidelines for Creating Clusters"](#) on page 5-14

If you decide to use hashing, then a table can still have separate indexes on any columns, including the cluster key.

**See Also:** For additional guidelines on the performance characteristics of hash clusters, see *Oracle8i Designing and Tuning for Performance*.

---

# Speeding Up Index Access with Index-Organized Tables

This chapter covers the following topics:

- [Overview of Index-Organized Tables](#)
- [Features of Index-Organized Tables](#)
- [When to Use Index-Organized Tables](#)
- [Example](#)

**See Also:** For the syntax of the ORGANIZATION INDEX clause of the CREATE TABLE statement, see *Oracle8i SQL Reference*.

## Overview of Index-Organized Tables

An index-organized table—in contrast to an ordinary table—has its own way of structuring, storing, and indexing data. A comparison with an ordinary table may help to explain its uniqueness.

### Index-Organized Tables versus Ordinary Tables

A row in an *ordinary table* has a stable physical location. Once it is given its first physical location, it never completely moves. Even if the row is partially moved with the addition of new data, there is always a row piece at the original physical address—identified by the original physical rowid—from which the system can find the rest of the row. As long as the row exists, its physical rowid does not change.

When you index a column in an ordinary table, the newly created index stores both the column data as well as the rowid.

A row in an *index-organized table* does *not* have a stable physical location. An index-organized table is, on the one hand, like an ordinary table with an index on one or more of its columns. It is unique, however, in that it holds its data, not in stable rows, but in sorted order in the leaves of a B\*-tree index built on the table's primary key. These rows may move around to retain the sorted order. An insertion, for example, can cause an index leaf to split and the existing row to be moved to a different slot, or even to a different block.

The leaves of the B\*-tree index hold the primary key and the actual row data. Changes to the table data—for example, adding new rows, or updating or deleting existing rows—result only in updating the index.

**See Also:** For more information on B\*-tree indexes, see *Oracle8i Concepts*

### Advantages of Index-Organized Tables

Because they store rows in the B\*-tree index based on the primary key, index-organized tables offer the following advantages over ordinary tables:

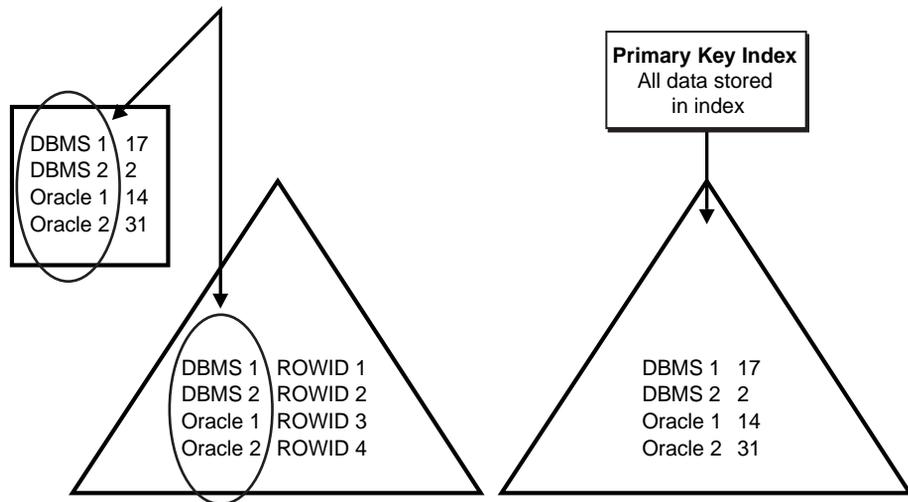
**Fast access to table data for queries involving exact match and/or range search on a primary key** Once a search has located the key values, the remaining data is present at that location. There is no need to follow a rowid back to table data, as would be the case with an ordinary table and index structure. The index-organized table thus shows its efficiency by eliminating one I/O, namely, the read of the table.

**Best table organization for 24x7 operations** When your database must be available 100% of the time, index-organized tables provide the following advantages:

- You can reorganize an index-organized table or an index-organized table partition (to recover space or improve performance) without rebuilding its indexes. This results in a short reorganization maintenance window.
- You can reorganize an index-organized table online. This and the ability to perform online reorganization of secondary indexes eliminates the reorganization maintenance window.

**Reduced storage requirements** This is because the key columns are not duplicated in both the table and the index, and because no additional storage is needed for rowids.

**Figure 6–1** Conventional Table and an Index versus Index-Organized Table



## Features of Index-Organized Tables

You can move your existing data into an index-organized table and do all the operations you would perform in an ordinary table. Some of the features now available to you in using index-organized tables include the following.

**Same Support for Alter Table Options as in Ordinary Tables** All of the alter options available on ordinary tables are now available for index-organized tables. This includes `ADD`, `MODIFY`, and `DROP COLUMNS` and `CONSTRAINTS`. However, the primary key constraint for an index-organized table cannot be dropped, deferred, or disabled.

**Logical ROWID Support** Because of the inherent movability of rows in a B\*-tree index, a secondary index on an index-organized table cannot be based on a *physical* rowid which is inherently fixed. Instead, a secondary index for an index-organized table is based on what is called the *logical* rowid. A logical rowid has no permanent physical address and can move across data blocks when new rows are inserted. However, if the physical location of a row changes, its logical rowid remains valid.

A logical rowid includes the table's primary key and a guess which identifies the block location of a row at the time the guess is made. The guess makes rowid access to non-volatile index-organized tables comparable to access of ordinary tables.

Logical rowids are similar to physical rowids in the following ways:

- Users can select ROWID from an index-organized table and access the rows using `WHERE ROWID = <value predicate>`.
- The access through the logical rowid is the fastest possible way to get to a specific row, even if it takes more than one block access to get it.
- Logical rowid of a row does not change as long as the primary key value does not change. However, unlike the physical rowid which remains immovable through all updates, the logical rowid can move when new rows are inserted.

Oracle 8i, release 8.1, introduces a single datatype, called universal rowid, to support both logical and physical rowids.

Applications which use rowids today may, if using index-organized tables, have to change to use universal rowids, but the changes are simpler due to the availability of UROWID datatype. This allows applications to access logical and physical rowids in a unified manner.

**For more information:** See "[Declaring a Universal Rowid Datatype](#)" on page 6-11.

**Secondary Index Support** Secondary indexes on index-organized tables differ from indexes on ordinary tables in two ways:

- They store logical rowids instead of physical rowids. Thus, a table maintenance operation such as `ALTER TABLE MOVE` does not make the secondary index unusable.
- The logical rowid also includes a guess that provides a direct access to the primary key index leaf block. If the guess is correct, a secondary index scan would incur a single additional I/O once the secondary key is found. The performance would be similar to that of a secondary index-scan on an ordinary table.

Both unique and non-unique secondary indexes, as well as function-based secondary indexes, are supported. However, bit-mapped secondary indexes on index-organized tables are currently not supported.

**LOB Columns** You can create internal and external LOB columns in index-organized tables to store large unstructured data such as audio, video, and images. The SQL DDL, DML, and piece-wise operations on LOBs in index-organized tables exhibit the same behavior as in ordinary tables. The main differences are:

- **Tablespace mapping**—By default (or unless specified otherwise), the LOB's data and index segments are created in the tablespace in which the primary key index segment of the index-organized table is created.
- **Inline vs. Out-of-line storage**—By default, all LOBs in index-organized tables created without an overflow segment are stored out-of-line (that is, the default storage attribute is `DISABLE STORAGE IN ROW`). Specifying an `ENABLE STORAGE IN ROW` for such LOBs, will result in an error. However, LOBs in index-organized tables with overflow segments have the same characteristics as those in ordinary tables.

Other LOB features—such as `BFILES`, temporary LOBs, and varying character width LOBs—are also supported in index-organized tables. You use them as you would ordinary tables. Support for LOBs in partitioned index-organized tables is not currently available.

**Parallel Query** Queries on index-organized tables involving primary key index scan can be executed in parallel. However, parallel execution of secondary index-only scan queries is not yet supported.

**Object Support** Most of the object features are supported on index-organized tables, including Object Type, `VARRAYs`, Nested Table, and REF Columns. However, you cannot create object tables (`TABLE OF <object type>`) as index-organized tables.

**SQL\*Loader** This utility supports both ordinary and direct path load of index-organized tables and their associated indexes (including partitioning support). However, direct path parallel load to an index-organized table is not supported. An alternate method of achieving the same result is to perform parallel load to an ordinary table using SQL\*Loader, then use the parallel `CREATE TABLE AS SELECT` option to build the index-organized table.

**Export/Import** This utility supports export (both ordinary and direct path) and import of non-partitioned and partitioned index-organized tables.

**Distributed Database and Replication Support** You can replicate both non-partitioned and partitioned index-organized tables.

**Tools** The Oracle Enterprise Manager supports generating SQL statements for `CREATE` and `ALTER` operations on an index-organized table.

**Key Compression** Key compression allows elimination of repeated occurrences of key column prefixes in index-organized tables and indexes. The salient characteristics of the scheme are:

- Key compression breaks an index key into a prefix entry and suffix entry. Compression is achieved by sharing the prefix entries among all the suffix entries in an index block.
- Only keys in the leaf blocks of a B\*-tree are compressed. Keys in the branch blocks of a B\*-tree are still suffix truncated but not subjected to key compression.

## When to Use Index-Organized Tables

There are several occasions when you may prefer to use index-organized tables over ordinary tables.<sup>1</sup>

**When You Want to Avoid Redundant Data Storage** For tables, where the majority of columns form the primary key, there is a significant amount of redundant data stored. You can avoid this redundant storage by using an index-organized table. Also, by using an index-organized table, you increase the efficiency of the primary key-based access to non-key columns.

**When Developing VLDB and OLTP Applications** The ability to partition an index-organized table on a range of column values makes the use of index-organized tables suitable for VLDB applications.

One major advantage of an index-organized table over an ordinary table stems from the logical nature of the index-organized table's secondary indexes. After an `ALTER TABLE MOVE` and `SPLIT` operation, *global* indexes on index-organized tables remain usable because the index rows contain logical rowids. In the case of ordinary tables, by contrast, these operations result in making the global index unusable, requiring a complete index rebuild, which can be very expensive.

Similarly, after an `ALTER TABLE MOVE` operation, *local* indexes on index-organized tables are still usable. On the other hand, for ordinary tables, the `MOVE` operation results in making a secondary local index unusable.

The partition maintenance operations described above do make the local and global indexes on index-organized table perform slower as the guess component of logical rowid becomes invalid. However, the indexes are still usable via the primary key-component of the logical rowid.

In addition, the `ALTER TABLE MOVE` operation can be done on-line. This functionality makes index-organized tables ideal for applications requiring 24X7 availability.

**When Developing Time-series Applications** The ability to cluster rows based on the primary key makes index-organized tables attractive for time-series

---

<sup>1</sup> If you use Oracle Advanced Queuing, you may be familiar with index-organized tables already. Oracle Advance Queuing provides message queuing as an integrated part of the Oracle8i server, and uses index-organized tables to hold metadata information for multiple consumer queues. In this case, the index-organized table acts as an "index," storing queue metadata as part of a primary key B\*-tree index on the queue identifier. DML operations in turn have to update the "index," and this occurs efficiently by updating the underlying index-organized table.

applications. Typically, a time-series is a set of time-stamped rows belonging to a single item such as stock price. Data is typically accessed through an item identifier such as a stock symbol and a time stamp. By defining, an index-organized table with primary key (stock symbol, time stamp), the Oracle8 Time Series Data Cartridge is able to store and manipulate time-series data efficiently. You can achieve further storage savings by compressing repeated occurrences of the item identifier (for example, the stock symbol) in a time series by using an index-organized table with key compression.

**When Using Nested Tables** For a nested table column, Oracle internally creates a storage table to hold all the nested table rows. The rows belonging to a single nested table instance are identified by a `NESTED_TABLE_ID` column. If an ordinary table is used as nested table column storage, the nested table rows typically get de-clustered. By contrast, when you use an index-organized table, the nested table rows can be clustered based on the `NESTED_TABLE_ID` column. In Oracle 8i, Release 8.1, you can specify the storage of the nested table to be an index-organized table, as illustrated in the following.

```
CREATE TYPE Project_t AS OBJECT(Pno NUMBER, Pname VARCHAR2(80));
CREATE TYPE Project_set AS TABLE OF Project_t;
CREATE TABLE Employees (Eno NUMBER, Projects PROJECT_SET)
  NESTED TABLE Projects_ntab STORE AS Emp_project_tab
  ((PRIMARY KEY(Nested_table_id, Pno)) ORGANIZATION INDEX)
  RETURN AS LOCATOR;
```

**When Using Extensible Indexing** Oracle 8i, Release 8.1 introduces the Extensible Indexing Framework which allows you to add a new access method to the database. Typically, domain-specific indexing schemes need some storage mechanism to hold their index data. Index-organized tables are ideal candidates for such domain index storage. Oracle8 Spatial and Text Database Cartridges have implemented domain-specific indexing schemes that use index-organized tables for storing their index data.

## Example

---

**Note:** You may need to set up the following data structures for certain examples to work; such as:

```
CONNECT system/manager
GRANT CREATE TABLESPACE TO scott;
CONNECT scott/tiger
CREATE TABLESPACE Ind_tbs DATAFILE 'disk1:moredata2'
SIZE 100K;
CREATE TABLE Doc_tab DATAFILE 'disk1:moredata2' SIZE
100K;
CREATE TABLESPACE Ovf_tbs DATAFILE 'disk1:moredata3'
SIZE 100K;
CREATE TABLESPACE Ind_ts0 DATAFILE 'disk1:moredata5'
SIZE 100K REUSE;
CREATE TABLESPACE Ov_ts0 DATAFILE 'disk1:moredata6'
SIZE 100K REUSE;
CREATE TABLESPACE Ind_ts1 DATAFILE 'disk1:moredata7'
SIZE 100K REUSE;
CREATE TABLESPACE Ov_ts1 DATAFILE 'disk1:moredata8'
SIZE 100K REUSE;
CREATE TABLESPACE Ind_ts2 DATAFILE 'disk1:moredata9'
SIZE 100K REUSE;
CREATE TABLESPACE Ov_ts2 DATAFILE 'disk1:moredata10'
SIZE 100K REUSE;
CREATE TABLE Doc_tab (tok VARCHAR2(4),id
VARCHAR2(14),freq NUMBER);
```

---

This example illustrates some of the basic tasks in creating and using index-organized tables. In this example, a text search engine uses an inverted index<sup>1</sup> to allow a user to query for specific words or phrases over the Web. It then returns to the user a list of hypertext links to documents containing the queried words and phrases, and it ranks those links in the order of relevance.

This example illustrates the following tasks:

---

<sup>1</sup> An inverted index breaks each document into individual words or tokens. For each word, the inverted index builds a list of documents in which the word occurs, then stores that list in the database. The application performs a content-based search by scanning the inverted index looking for tokens of interest. From a development standpoint, an inverted index typically contains entries of the form <token, document\_id, occurrence\_data> for each distinct word in a document.

- [Moving Existing Data from an Ordinary Table into an Index-Organized Table](#)
- [Creating Index-Organized Tables](#)
- [Declaring a Universal Rowid Datatype](#)
- [Creating Secondary Indexes on Index-Organized Tables](#)
- [Manipulating Index-Organized Tables](#)
- [Specifying an Overflow Data Segment](#)
- [Determining the Last Non-key Column Included in the Index Row Head Piece](#)
- [Storing Columns in the Overflow Segment](#)
- [Modifying Physical and Storage Attributes](#)
- [Partitioning an Index-Organized Table](#)
- [Rebuilding an Index-Organized Table](#)

### **Moving Existing Data from an Ordinary Table into an Index-Organized Table**

The `CREATE TABLE AS SELECT` command allows you to move existing data from an ordinary table into an index-organized table. In the following example, an index-organized table, called `docindex`, is created from an ordinary table called `doctable`.

```
CREATE TABLE Docindex
  (   Token,
      Doc_id,
      Token_frequency,
      CONSTRAINT Pk_docindex PRIMARY KEY (Token, Doc_id)
  )
ORGANIZATION INDEX TABLESPACE Ind_tbs
PARALLEL (DEGREE 2)
AS SELECT * from Doc_tab;
```

Note that the `PARALLEL` clause allows the table creation to be performed in parallel.

### **Creating Index-Organized Tables**

To create an index-organized table, you use the `ORGANIZATION INDEX` clause. In the following example, an inverted index—typically used by Web text-search engines—uses an index-organized table.

```
CREATE TABLE Docindex
  (   Token          CHAR(20),
```

```

Doc_id          NUMBER,
Token_frequency NUMBER,
CONSTRAINT Pk_docindex PRIMARY KEY (Token, Doc_id)
)
ORGANIZATION INDEX TABLESPACE Ind_tbs;

```

## Declaring a Universal Rowid Datatype

The following example shows how you declare the UROWID datatype.

```

DECLARE
  Rid UROWID;
BEGIN
  INSERT INTO Docindex VALUES ('Or80', 2, 30)
    RETURNING Rowid INTO RID;
  UPDATE Docindex SET Token='Or81' WHERE ROWID = Rid;
END;

```

## Creating Secondary Indexes on Index-Organized Tables

You can create secondary indexes on index-organized tables to provide multiple access paths. The following example shows the creation of an index on (doc\_id, token).

```

CREATE INDEX Doc_id_index on Docindex(Doc_id, Token);

```

This secondary index allows Oracle to efficiently process queries involving predicates on doc\_id, as the following example illustrates.

```

SELECT Token FROM Docindex WHERE Doc_id = 1;

```

## Manipulating Index-Organized Tables

Applications manipulate the index-organized tables just like an ordinary table, using standard SQL statements for SELECT, INSERT, UPDATE, or DELETE operations. For example, you can manipulate the docindex table as follows:

```

INSERT INTO Docindex VALUES ('Oracle8.1', 3, 17);
SELECT * FROM Docindex;
UPDATE Docindex SET Token = 'Oracle8' WHERE Token = 'Oracle8.1';
DELETE FROM Docindex WHERE Doc_id = 1;

```

Also, you can use SELECT FOR UPDATE statements to lock rows of an index-organized table. All of these operations result in manipulating the primary key B\*-tree index. Both query and DML operations involving index-organized tables are optimized by using this cost-based approach.

## Specifying an Overflow Data Segment

Storing all non-key columns in the primary key B\*-tree index structure may not always be desirable because, for example:

- Each additional non-key column stored in the primary key index reduces the dense clustering of index rows in the B\*-tree index leaf blocks  
or because
- A leaf block of a B\*-tree must hold at least two index rows, and putting all non-key columns as part of an index row may not be possible.

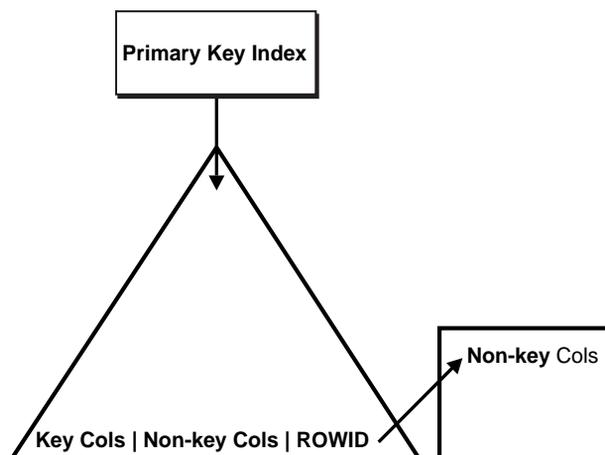
To overcome these problems, you can associate an overflow data segment with an index-organized table. In the following example, an additional column, `token_offsets`, is required for the `docindex` table. This example shows how you can create an index-organized table and use the `OVERFLOW` option to create an overflow data segment.

```
CREATE TABLE Docindex2
(   Token          CHAR(20),
    Doc_id         NUMBER,
    Token_frequency NUMBER,
    Token_offsets  VARCHAR(512),
    CONSTRAINT Pk_docindex2 PRIMARY KEY (Token, Doc_id)
)
ORGANIZATION INDEX TABLESPACE Ind_tbs PCTTHRESHOLD 20
OVERFLOW TABLESPACE Ovf_tbs INITRANS 4;
```

For the overflow data segment, you can specify physical storage attributes such as `TABLESPACE`, `INITRANS`, and so on.

For an index-organized table with an overflow segment, the index row contains a `<key, row head>` pair, where the row head contains the first few non-key columns and a rowid that points to an overflow row-piece containing the remaining column values. Although this approach incurs the storage cost of one rowid per row, it nevertheless avoids key column duplication.

**Figure 6–2 Overflow Segment**



### Determining the Last Non-key Column Included in the Index Row Head Piece

To determine the last non-key column to include in the index row head piece, you use the `PCTTHRESHOLD` option specified as a percentage of the leaf block size. The remaining non-key columns are stored in the overflow data segment as one or more row-pieces. Specifically, the last non-key column to be included is chosen so that the index row size (key + row head) does not exceed the specified threshold (which, in the following example, is 20% of the index leaf block). By default, `PCTTHRESHOLD` is set at 50 when omitted.

The `PCTTHRESHOLD` option determines the last non-key column to be included in the index on a per row basis. It does not, however, allow you to specify that the same set of columns be included in the index for all rows in the table. For this purpose, the `INCLUDING` option is provided.

The `CREATE TABLE` statement in the following example includes all the columns up to the `token_frequency` column in the index leaf block and forces the `token_offsets` column to the overflow segment.

```
CREATE TABLE Docindex3
(
  Token          CHAR(20),
  Doc_id        NUMBER,
  Token_frequency NUMBER,
  Token_offsets  VARCHAR(512),
  CONSTRAINT Pk_docindex3 PRIMARY KEY (Token, Doc_id)
)
ORGANIZATION INDEX TABLESPACE Ind_tbs INCLUDING Token_frequency
```

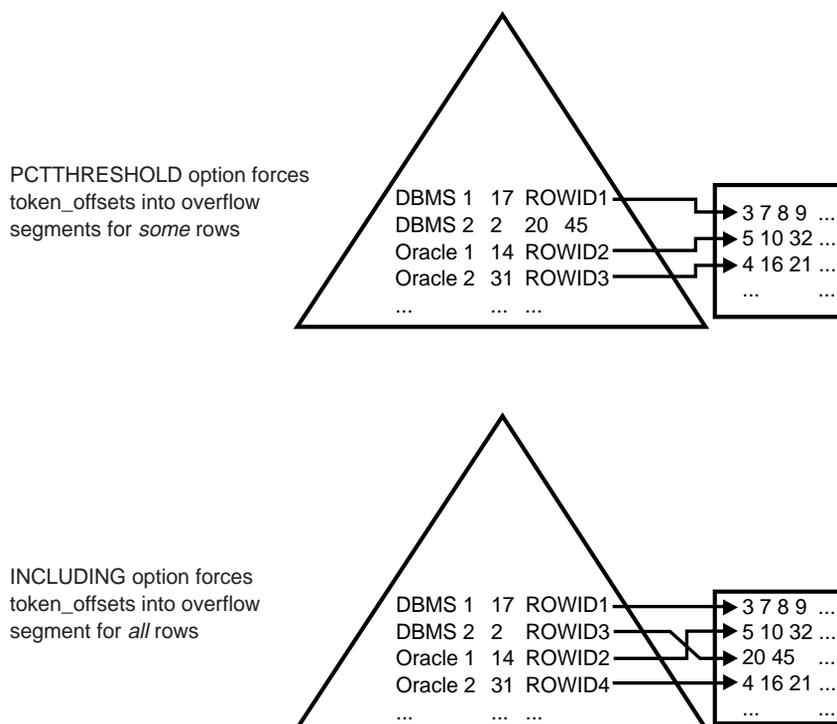
```
OVERFLOW TABLESPACE Ovf_tbs;
```

Such vertical partitioning of a row between the index and data segments allows for higher clustering of rows in the index. This results in better query performance for the columns stored in the index. For example, if the `token_offsets` column is infrequently accessed, then pushing this column out of the index results in better clustering of index rows in the primary key B\*-tree structure (Figure 6-3). This in turn results in overall improved query performance. However, there is one additional block access for columns stored in the overflow data segment, and this can slow performance.

### Storing Columns in the Overflow Segment

The `INCLUDING` option ensures that all columns after the specified including column are stored in the overflow segment. If the including column specified is such that corresponding index row size exceeds the specified threshold, then the last non-key column to be included is determined according to the `PCTTHRESHOLD` option.

**Figure 6–3 PCTTHRESHOLD versus INCLUDING Column Usage**



## Modifying Physical and Storage Attributes

You can use the `ALTER TABLE` command to modify physical and storage attributes for both the index and overflow data segments as well as alter `PCTTHRESHOLD` and `INCLUDING` column values. The following example sets the `INITRANS` of index segment to 4, `PCTTHRESHOLD` to 20, and the `INITRANS` of the overflow data segment to 6. The altered values are used for subsequent operations on the table.

```
ALTER TABLE Docindex INITRANS 4 PCTTHRESHOLD 20 OVERFLOW INITRANS 6;
```

For index-organized tables created without an overflow data segment, you can add an overflow data segment using `ALTER TABLE ADD OVERFLOW` option. The following example shows how to add an overflow segment to the `docindex` table.

```
ALTER TABLE Docindex ADD OVERFLOW;
```

## Analyzing an Index-Organized Table

Index-organized tables are analyzed just like ordinary tables using the `ANALYZE` command. The following example illustrates how you could use the `ANALYZE` command to analyze the `docindex` table.

```
ANALYZE TABLE Docindex COMPUTE STATISTICS;
```

Using the `ANALYZE` command analyzes both the primary key index segment and the overflow data segment, and computes logical as well as physical statistics for the table. Also, you can determine how many rows have one or more chained overflow row-pieces using the `ANALYZE LIST CHAINED ROWS` option. However, to identify the chain rows, you must create a slightly different `CHAINED_ROWS` table that includes primary key columns. With the logical rowid support added in Oracle8i, Release 8.1.5, a separate `CHAINED_ROWS` table is no longer needed.

## Loading, exporting/importing, replicating an Index-Organized Table

Data can be loaded into both non-partitioned and partitioned index-organized tables using the ordinary or direct path with the SQL\*Loader. The data can also be exported or imported using the Export/Import utility. Index-organized tables can also be replicated in a distributed database just like ordinary tables.

## Partitioning an Index-Organized Table

You can partition index-organized tables by range of column values. However, to create such partitioned index-organized tables *the set of partitioning columns must be a subset of primary key columns*. By imposing this restriction, only a single partition needs to be searched for to verify the uniqueness of the primary key during DML operations. This preserves the partition independence property.

The following are key aspects of partitioned index-organized tables:

- You must specify the `ORGANIZATION INDEX` clause to create an index-organized table as part of table-level attributes. This property is implicitly inherited by all partitions.
- You must specify the `OVERFLOW` option as part of table-level attribute to create an index-organized table with overflow data segment.
- The `OVERFLOW` option results in the creation of overflow data segments, which are themselves equi-partitioned with the primary key index segments. That is, each partition has an index segment and an overflow data segment.

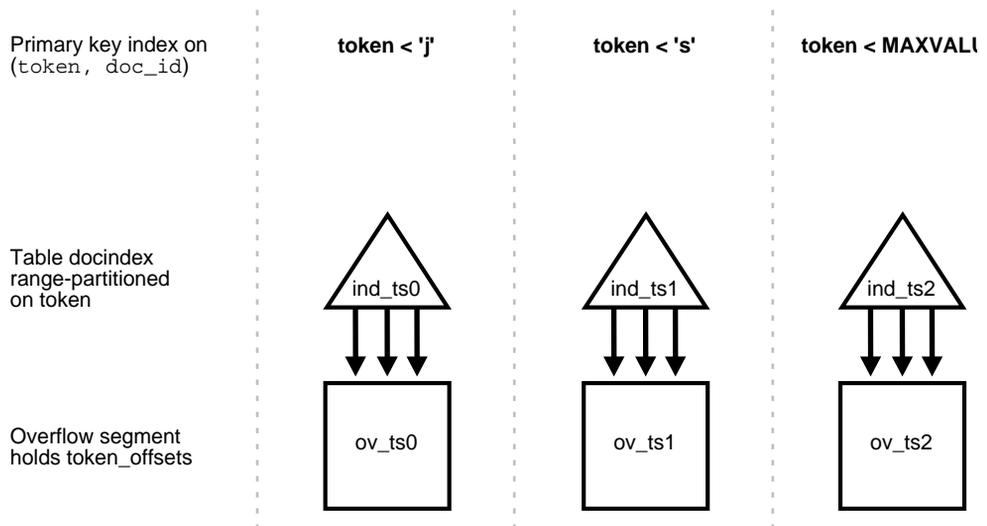
- As in ordinary partitioned tables, you can specify default values for physical attributes at the table-level. These can be overridden for each partition (both for index and overflow data segment).
- The tablespace for index segment, if not specified for a partition, is set to the table level default. If the table level default is not specified, then the default tablespace for the user is used.
- The default values for `PCTTHRESHOLD` and `INCLUDING` column can only be specified at the table-level.
- All the attributes that are specified before the `OVERFLOW` keyword are applicable to primary index segments. All the attributes specified after the `OVERFLOW` keyword are applicable to overflow data segments.
- The tablespace for an overflow data segment, if not specified for a partition, is set to the table level default. If the table-level default is not specified, then the tablespace of the corresponding partition's index segment is used.

The following example continues the example of the `docindex` table. It illustrates a range partition on token values.

```
CREATE TABLE Docindex4
  (Token          CHAR(20),
   Doc_id         NUMBER,
   Token_frequency NUMBER,
   Token_offsets  VARCHAR(512),
   CONSTRAINT Pk_docindex4 PRIMARY KEY (Token, Doc_id)
  )
  ORGANIZATION INDEX INITRANS 4 INCLUDING Token_frequency
  OVERFLOW INITRANS 6
  PARTITION BY RANGE(token)
  ( PARTITION P1 VALUES LESS THAN ('j')
    TABLESPACE Ind_ts0 OVERFLOW TABLESPACE Ov_ts0,
    PARTITION P2 VALUES LESS THAN ('s')
    TABLESPACE Ind_ts1 OVERFLOW TABLESPACE Ov_ts1,
    PARTITION P3 VALUES LESS THAN (MAXVALUE)
    TABLESPACE Ind_ts2 OVERFLOW TABLESPACE Ov_ts2);
```

This will result in creation of the table shown in [Figure 6-4](#). The `INCLUDING` column results in storing the `token_offsets` in the overflow data segment for each partition.

**Figure 6–4 Range-partitioned Index-organized Table with Overflow Segment**



Support for partitioned indexes on index-organized tables is very similar to that for an ordinary table. Local prefixed, local non-prefixed, and global prefixed partitioned indexes are supported on index-organized tables. The only difference is that these indexes store logical rowids instead of physical rowids.

All of the ALTER TABLE operations, except MERGE, are available for partitioned index-organized tables. However, there are some differences in behavior with respect to ordinary tables:

- For ALTER TABLE MOVE partition operations, *all indexes—local, global, and non-partitioned—remain* **USABLE** because the indexes contain logical rowids. However, the guess stored in the logical rowid becomes invalid.
- For SPLIT partition operations, all indexes or global index partitions remain usable.
- For ALTER TABLE EXCHANGE partition, the target table must be a compatible index-organized table.
- Users can use the ALTER TABLE ADD OVERFLOW command to add an overflow segment and specify table-level default and partition-level physical and storage attributes. This operation results in adding an overflow data segment to each partition.

ALTER INDEX operations are very similar to those on ordinary tables. The only difference is that operations that reconstruct the entire index—namely, ALTER INDEX REBUILD and SPLIT\_PARTITION—result in reconstructing the guess stored as part of the logical rowid.

Query and DML operations on partitioned index-organized tables work the same as on ordinary partitioned tables.

## Key Compression

You enable key compression by using the COMPRESS clause when specifying physical attributes for the index segment. In addition, the prefix length (as number of columns) can be specified to identify how the key can be broken into a prefix and a suffix. The valid range of values for prefix length are [1, number of primary key columns minus 1].

```
CREATE TABLE Docindex5
  ( Token          CHAR(20),
    Doc_id         NUMBER,
    Token_frequency NUMBER,
    Token_offsets  VARCHAR(512),
    CONSTRAINT pk_docindex5 PRIMARY KEY (Token, Doc_id)
  )
  ORGANIZATION INDEX TABLESPACE Ind_tbs COMPRESS 1 INCLUDING Token_frequency
  OVERFLOW TABLESPACE Ovf_tbs;
```

Common prefixes of length 1 (that is, token column) will be compressed in the primary key (token, doc\_id) occurrences. For the list of primary key values ('DBMS', 1), ('DBMS', 2), ('Oracle', 1), ('Oracle', 2), the repeated occurrences of 'DBMS' and 'Oracle' are compressed away.

If a prefix length is not specified, by default it is set to number of primary key columns minus 1. The compress option can be specified during creation of an index-organized table or as part of moving the index-organized table using ALTER TABLE MOVE option. For example, you can disable compression as follows:

```
ALTER TABLE Docindex5 MOVE NOCOMPRESS;
```

Similarly, the indexes for ordinary tables and index-organized tables can be compressed using the COMPRESS option.

**Key Compression for Partitioned Index-Organized Tables** Key compression is also supported for partitioned index-organized tables. The compression clause must be specified as part of table-level defaults. For each partition, compression can be

enabled or disabled. However, the prefix length cannot be changed at partition level.

```
CREATE TABLE Docindex6
  ( Token          CHAR(20),
    Doc_id         NUMBER,
    Token_frequency NUMBER,
    Token_offsets  VARCHAR(512),
    CONSTRAINT Pk_docindex6 PRIMARY KEY (Token, Doc_id)
  )
ORGANIZATION INDEX INITRANS 4 COMPRESS 1 INCLUDING Token_frequency
OVERFLOW INITRANS 6
      PARTITION BY RANGE(Token)
      ( PARTITION P1 VALUES LESS THAN ('j')
        TABLESPACE Ind_ts0 OVERFLOW TABLESPACE Ov_ts0,
        PARTITION P2 VALUES LESS THAN ('s')
        TABLESPACE Ind_ts1 NOCOMPRESS OVERFLOW TABLESPACE Ov_ts1,
        PARTITION P3 VALUES LESS THAN (MAXVALUE)
        TABLESPACE Ind_ts2 OVERFLOW TABLESPACE Ov_ts2
      );
```

All partitions inherit the table-level default for prefix length. Partitions P1 and P3 are created with key-compression enabled. For partition P2, the compression is disabled by the partition level NOCOMPRESS option.

For ALTER TABLE MOVE and SPLIT operations, the COMPRESS option can be altered. The following example rebuilds the partition with key compression enabled.

```
ALTER TABLE Docindex6 MOVE PARTITION P2 COMPRESS;
```

### Rebuilding an Index-Organized Table

A new SQL command, ALTER TABLE MOVE, allows you to move, that is, rebuild the table. This should be used when the B\*-tree structure containing an index-organized table gets fragmented due to a large number of inserts, updates, or deletes. The MOVE option rebuilds the primary key B\*-tree index.

By default, the overflow data segment is not rebuilt, except when:

- The OVERFLOW clause is explicitly specified,
- The PCTHRESHOLD and/or INCLUDING column value are altered as part of the MOVE statement.
- Any LOBs are moved explicitly

---

By default, LOB columns related to index and data segments are not rebuilt, except when the LOB columns are explicitly specified as part of the MOVE statement. The following example rebuilds the B\*-tree index containing the table data after setting the INITTRANS to 6 for index blocks.

```
ALTER TABLE docindex MOVE INITTRANS 6;
```

The following example rebuilds both the primary key index and overflow data segment.

```
ALTER TABLE docindex MOVE TABLESPACE Ovf_tbs OVERFLOW TABLESPACE ov_ts0;
```

By default, during the move, the table is not available for other operations. However, you can move an index-organized table using the ONLINE option. The following example allows the table to be available for DML and query operations during the actual move operation. This feature makes the index-organized table suitable for applications requiring 24X7 availability.

---

---

**Caution:** You may need to set your COMPATIBLE initialization parameter to '8.1.3.0' or higher to get the following to work:

---

---

```
ALTER TABLE Docindex MOVE ONLINE;
```

The MOVE option is also available for ordinary tables. However, ONLINE move is supported only for index-organized tables which do not have an overflow segment.



---

# Processing SQL Statements

This chapter describes how Oracle processes Structured Query Language (SQL) statements. Topics include the following:

- [SQL Statement Execution](#)
- [Controlling Transactions](#)
- [Ensuring Repeatable Reads with Read-Only Transactions](#)
- [Using Cursors](#)
- [Explicit Data Locking](#)
- [Explicitly Acquiring Row Locks](#)
- [Letting Oracle Control Table Locking](#)
- [User Locks](#)
- [Concurrency Control Using Serializable Transactions](#)
- [Autonomous Transactions](#)

Although some Oracle tools and applications simplify or mask the use of SQL, all database operations are performed using SQL. Any other data access method would circumvent the security built into Oracle and potentially compromise data security and integrity.

## SQL Statement Execution

**Table 7-1** outlines the stages commonly used to process and execute a SQL statement. In some cases, these steps might be executed in a slightly different order. For example, the `DEFINE` stage could occur just before the `FETCH` stage, depending on how your code is written.

For many Oracle tools, several of the stages are performed automatically. Most users do not need to be concerned with, or aware of, this level of detail. However, you might find this information useful when writing Oracle applications.

**See Also:** Refer to *Oracle8i Concepts* for a description of each stage of SQL statement processing for each type of SQL statement.

### Identifying Extensions to SQL92 (FIPS Flagging)

The Federal Information Processing Standard for SQL (FIPS 127-2) requires a way to identify SQL statements that use vendor-supplied extensions. Oracle provides a FIPS flagger to help you write portable applications.

When FIPS flagging is active, your SQL statements are checked to see whether they include extensions that go beyond the ANSI/ISO SQL92 standard. If any non-standard constructs are found, then the Oracle Server flags them as errors and displays the violating syntax.

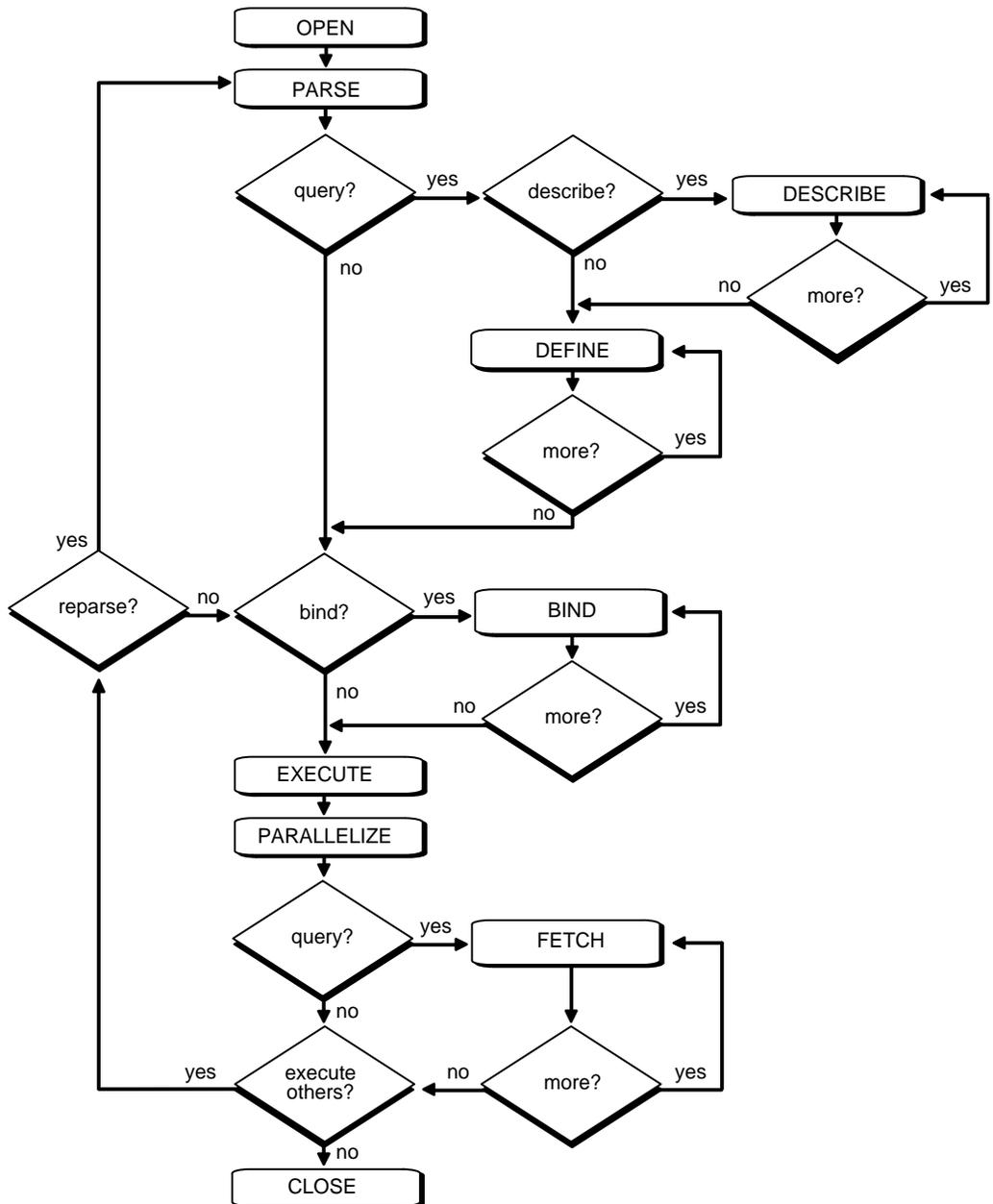
The FIPS flagging feature supports flagging through interactive SQL statements submitted using Enterprise Manager or SQL\*Plus. The Oracle Precompilers and SQL\*Module also support FIPS flagging of embedded and module language SQL.

When flagging is on and non-standard SQL is encountered, the following message is returned:

```
ORA-00097: Use of Oracle SQL feature not in SQL92 level Level
```

Where *level* can be either `ENTRY`, `INTERMEDIATE`, or `FULL`.

Figure 7-1 The Stages in Processing a SQL Statement



## Controlling Transactions

In general, only application designers using the programming interfaces to Oracle are concerned with which types of actions should be grouped together as one transaction. Transactions must be defined properly so work is accomplished in logical units and data is kept consistent. A transaction should consist of all of the necessary parts for one logical unit of work, no more and no less. Data in all referenced tables should be in a consistent state before the transaction begins and after it ends. Transactions should consist of only the SQL statements or PL/SQL blocks that comprise one consistent change to the data.

A transfer of funds between two accounts (the transaction or logical unit of work), for example, should include the debit to one account (one SQL statement) and the credit to another account (one SQL statement). Both actions should either fail or succeed together as a unit of work; the credit should not be committed without the debit. Other non-related actions, such as a new deposit to one account, should not be included in the transfer of funds transaction.

## Improving Performance

In addition to determining which types of actions form a transaction, when you design an application, you must also determine if you can take any additional measures to improve performance. You should consider the following performance enhancements when designing and writing your application. Unless otherwise noted, each of these features is described in *Oracle8i Concepts*.

- Use the `BEGIN_DISCRETE_TRANSACTION` procedure to improve the performance of short, non-distributed transactions.
- Use the `SET TRANSACTION` command with the `USE_ROLLBACK_SEGMENT` parameter to explicitly assign a transaction to an appropriate rollback segment. This can eliminate the need to dynamically allocate additional extents, which can reduce overall system performance.
- Use the `SET TRANSACTION` command with the `ISOLATION_LEVEL` set to `SERIALIZABLE` to get ANSI/ISO serializable transactions.

### See Also:

- ["Serializable Transaction Interaction"](#) on page 7-25
- *Oracle8i Concepts*.
- Establish standards for writing SQL statements so that you can take advantage of shared SQL areas. Oracle recognizes identical SQL statements and allows

them to share memory areas. This reduces memory storage usage on the database server, thereby increasing system throughput.

- Use the `ANALYZE` command to collect statistics that can be used by Oracle to implement a cost-based approach to SQL statement optimization. You can supply additional "hints" to the optimizer as needed.
- Call the `DBMS_APPLICATION_INFO.SET_ACTION` procedure before beginning a transaction to register and name a transaction for later use when measuring performance across an application. You should specify what type of activity a transaction performs so that the system tuners can later see which transactions are taking up the most system resources.
- Increase user productivity and query efficiency by including user-written PL/SQL functions in SQL expressions as described in "Calling Stored Functions from SQL Expressions".
- Create explicit cursors when writing a PL/SQL application.
- When writing precompiler programs, increasing the number of cursors using `MAX_OPEN_CURSORS` can often reduce the frequency of parsing and improve performance.

**See Also:** "Using Cursors" on page 7-9

## Committing a Transaction

To commit a transaction, use the `COMMIT` command. The following two statements are equivalent and commit the current transaction:

```
COMMIT WORK;  
COMMIT;
```

The `COMMIT` command allows you to include the `COMMENT` parameter along with a comment (less than 50 characters) that provides information about the transaction being committed. This option is useful for including information about the origin of the transaction when you commit distributed transactions:

```
COMMIT COMMENT 'Dallas/Accts_pay/Trans_type 10B';
```

**See Also:** For additional information about committing in-doubt distributed transactions, see *Oracle8 Distributed Database Systems*.

## Rolling Back a Transaction

To roll back an entire transaction or a part of a transaction (that is, to a savepoint), use the `ROLLBACK` command. For example, either of the following statements rolls back the entire current transaction:

```
ROLLBACK WORK;  
ROLLBACK;
```

The `WORK` option of the `ROLLBACK` command has no function.

To roll back to a savepoint defined in the current transaction, the `TO` option of the `ROLLBACK` command must be used. For example, either of the following statements rolls back the current transaction to the savepoint named `POINT1`:

```
SAVEPOINT Point1;  
...  
ROLLBACK TO SAVEPOINT Point1;  
ROLLBACK TO Point1;
```

**See Also:** For additional information about rolling back in-doubt distributed transactions, see *Oracle8 Distributed Database Systems*.

## Defining a Transaction Savepoint

To define a *savepoint* in a transaction, use the `SAVEPOINT` command. The following statement creates the savepoint named `ADD_EMP1` in the current transaction:

```
SAVEPOINT Add_emp1;
```

If you create a second savepoint with the same identifier as an earlier savepoint, then the earlier savepoint is erased. After a savepoint has been created, you can roll back to the savepoint.

There is no limit on the number of active savepoints per session. An active savepoint is one that has been specified since the last commit or rollback.

### An Example of COMMIT, SAVEPOINT, and ROLLBACK

The following series of SQL statements illustrates the use of `COMMIT`, `SAVEPOINT`, and `ROLLBACK` statements within a transaction:

SQL Statement	Results
SAVEPOINT a;	First savepoint of this transaction
DELETE...;	First DML statement of this transaction
SAVEPOINT b;	Second savepoint of this transaction
INSERT INTO...;	Second DML statement of this transaction
SAVEPOINT c;	Third savepoint of this transaction
UPDATE...;	Third DML statement of this transaction.
ROLLBACK TO c;	UPDATE statement is rolled back, savepoint C remains defined
ROLLBACK TO b;	INSERT statement is rolled back, savepoint C is lost, savepoint B remains defined
ROLLBACK TO c;	ORA-01086 error; savepoint C no longer defined
INSERT INTO...;	New DML statement in this transaction
COMMIT;	Commits all actions performed by the first DML statement (the DELETE statement) and the last DML statement (the second INSERT statement)  All other statements (the second and the third statements) of the transaction had been rolled back before the COMMIT. The savepoint A is no longer active.

## Privileges Required for Transaction Management

No privileges are required to control your own transactions; any user can issue a COMMIT, ROLLBACK, or SAVEPOINT statement within a transaction.

## Ensuring Repeatable Reads with Read-Only Transactions

By default, the consistency model for Oracle guarantees statement-level read consistency, but does not guarantee transaction-level read consistency (repeatable reads). If you want transaction-level read consistency, and if your transaction does not require updates, then you can specify a *read-only transaction*. After indicating that your transaction is read-only, you can execute as many queries as you like against any database table, knowing that the results of each query in the read-only transaction are consistent with respect to a single point in time.

A read-only transaction does not acquire any additional data locks to provide transaction-level read consistency. The multi-version consistency model used for statement-level read consistency is used to provide transaction-level read consistency; all queries return information with respect to the system control number (SCN) determined when the read-only transaction begins. Because no data locks are acquired, other transactions can query and update data being queried concurrently by a read-only transaction.

Changed data blocks queried by a read-only transaction are reconstructed using data from rollback segments. Therefore, long running read-only transactions sometimes receive a "snapshot too old" error (ORA-01555). Create more, or larger, rollback segments to avoid this. Alternatively, you could issue long-running queries when online transaction processing is at a minimum, or you could obtain a shared lock on the table you were querying, prohibiting any other modifications during the transaction.

A read-only transaction is started with a `SET TRANSACTION` statement that includes the `READ ONLY` option. For example:

```
SET TRANSACTION READ ONLY;
```

The `SET TRANSACTION` statement must be the first statement of a new transaction; if any DML statements (including queries) or other non-DDL statements (such as `SET ROLE`) precede a `SET TRANSACTION READ ONLY` statement, then an error is returned. Once a `SET TRANSACTION READ ONLY` statement successfully executes, only `SELECT` (without a `FOR UPDATE` clause), `COMMIT`, `ROLLBACK`, or non-DML statements (such as `SET ROLE`, `ALTER SYSTEM`, `LOCK TABLE`) are allowed in the transaction. Otherwise, an error is returned. A `COMMIT`, `ROLLBACK`, or DDL statement terminates the read-only transaction (a DDL statement causes an implicit commit of the read-only transaction and commits in its own transaction).

## Using Cursors

PL/SQL implicitly declares a cursor for all SQL data manipulation statements, including queries that return only one row. For queries that return more than one row, you can explicitly declare a cursor to process the rows individually.

A *cursor* is a handle to a specific private SQL area. In other words, a cursor can be thought of as a name for a specific private SQL area. A PL/SQL *cursor variable* enables the retrieval of multiple rows from a stored procedure. Cursor variables allow you to pass cursors as parameters in your 3GL application. Cursor variables are described in *PL/SQL User's Guide and Reference*.

Although most Oracle users rely on the automatic cursor handling of the Oracle utilities, the programmatic interfaces offer application designers more control over cursors. In application development, a cursor is a named resource available to a program, which can be specifically used for parsing SQL statements embedded within the application.

## Declaring and Opening Cursors

There is no absolute limit to the total number of cursors one session can have open at one time, subject to two constraints:

- Each cursor requires virtual memory, so a session's total number of cursors is limited by the memory available to that process.
- A system-wide limit of cursors per session is set by the initialization parameter named `OPEN_CURSORS` found in the parameter file (such as `INIT.ORA`).

**See Also:** Parameters are described in *Oracle8i Reference*.

Explicitly creating cursors for precompiler programs can offer some advantages in tuning those applications. For example, increasing the number of cursors can often reduce the frequency of parsing and improve performance. If you know how many cursors may be required at a given time, then you can make sure you can open that many simultaneously.

## Using a Cursor to Re-Execute Statements

After each stage of execution, the cursor retains enough information about the SQL statement to re-execute the statement without starting over, as long as no other SQL statement has been associated with that cursor. This is illustrated in [Figure 7-1](#). Notice that the statement can be re-executed without including the parse stage.

By opening several cursors, the parsed representation of several SQL statements can be saved. Repeated execution of the same SQL statements can thus begin at the describe, define, bind, or execute step, saving the repeated cost of opening cursors and parsing.

## Closing Cursors

Closing a cursor means that the information currently in the associated private area is lost and its memory is deallocated. Once a cursor is opened, it is not closed until one of the following events occurs:

- The user program terminates its connection to the server.
- If the user program is an OCI program or precompiler application, then it explicitly closes any open cursor during the execution of that program. (However, when this program terminates, any cursors remaining open are implicitly closed.)

## Cancelling Cursors

Cancelling a cursor frees resources from the current fetch. The information currently in the associated private area is lost but the cursor remains open, parsed, and associated with its bind variables.

---

---

**Note:** You cannot cancel cursors using Pro\*C or PL/SQL.

---

---

**See Also:** For more information about cancelling cursors, see *Oracle Call Interface Programmer's Guide*.

## Explicit Data Locking

Oracle always performs necessary locking to ensure data concurrency, integrity, and statement-level read consistency. However, options are available to override the default locking mechanisms. Situations where it would be advantageous to override the default locking of Oracle include the following:

- An application desires transaction-level read consistency or "repeatable reads"—transactions must query a consistent set of data for the duration of the transaction, knowing that the data has not been changed by any other transactions of the system. Transaction-level read consistency can be achieved by using explicit locking, read-only transactions, serializable transactions, or overriding default locking for the system.
- An application requires a transaction to have exclusive access to a resource. To proceed with its statements, the transaction with exclusive access to a resource does not have to wait for other transactions to complete.

The automatic locking mechanisms can be overridden at two different levels:

transaction level	Transactions including the following SQL statements override Oracle's default locking: the <code>LOCK TABLE</code> command, the <code>SELECT</code> command including the <code>FOR UPDATE</code> clause, and the <code>SET TRANSACTION</code> command with the <code>READ ONLY</code> or <code>ISOLATION LEVEL SERIALIZABLE</code> options. Locks acquired by these statements are released after the transaction is committed or rolled back.
system level	An instance can be started with non-default locking by adjusting the initialization parameters <code>SERIALIZABLE</code> and <code>ROW_LOCKING</code> .

The following sections describe each option available for overriding the default locking of Oracle. The initialization parameter `DML_LOCKS` determines the maximum number of DML locks allowed.

**See Also:** See the *Oracle8i Reference* for a discussion of parameters.

The default value should be sufficient; however, if you are using additional manual locks, then you may need to increase this value.

---

---

**Caution:** If you override the default locking of Oracle at any level, then be sure that the overriding locking procedures operate correctly: Ensure that data integrity is guaranteed, data concurrency is acceptable, and deadlocks are not possible or are appropriately handled.

---

---

## Choosing a Locking Strategy

A transaction explicitly acquires the specified table locks when a `LOCK TABLE` statement is executed. A `LOCK TABLE` statement manually overrides default locking. When a `LOCK TABLE` statement is issued on a view, the underlying base tables are locked. The following statement acquires exclusive table locks for the `EMP_TAB` and `DEPT_TAB` tables on behalf of the containing transaction:

```
LOCK TABLE Emp_tab, Dept_tab
  IN EXCLUSIVE MODE NOWAIT;
```

You can specify several tables or views to lock in the same mode; however, only a single lock mode can be specified per `LOCK TABLE` statement.

---

---

**Note:** When a table is locked, all rows of the table are locked. No other user can modify the table.

---

---

You can also indicate if you do or do not want to wait to acquire the lock. If you specify the `NOWAIT` option, then you only acquire the table lock if it is immediately available. Otherwise an error is returned to notify that the lock is not available at this time. In this case, you can attempt to lock the resource at a later time. If `NOWAIT` is omitted, then the transaction does not proceed until the requested table lock is acquired. If the wait for a table lock is excessive, then you might want to cancel the lock operation and retry at a later time; you can code this logic into your applications.

---

---

**Note:** A distributed transaction waiting for a table lock can time-out waiting for the requested lock if the elapsed amount of time reaches the interval set by the initialization parameter `DISTRIBUTED_LOCK_TIMEOUT`. Because no data has been modified, no actions are necessary as a result of the time-out. Your application should proceed as if a deadlock has been encountered. For more information on distributed transactions, refer to *Oracle8 Distributed Database Systems*.

---

---

The following paragraphs provide guidance on when it can be advantageous to acquire each type of table lock using the `LOCK TABLE` command.

### ROW SHARE and ROW EXCLUSIVE

```
LOCK TABLE Emp_tab IN ROW SHARE MODE;  
LOCK TABLE Emp_tab IN ROW EXCLUSIVE MODE;
```

Row share and row exclusive table locks offer the highest degree of concurrency. Conditions that possibly warrant the explicit acquisition of a row share or row exclusive table lock include the following:

- Your transaction needs to prevent another transaction from acquiring an intervening share, share row, or exclusive table lock for a table before the table can be updated in your transaction. If another transaction acquires an intervening share, share row, or exclusive table lock, then no other transactions can update the table until the locking transaction commits or rolls back.
- Your transaction needs to prevent a table from being altered or dropped before the table can be modified later in your transaction.

### SHARE

```
LOCK TABLE Emp_tab IN SHARE MODE;
```

Share table locks are rather restrictive data locks. The following conditions could warrant the explicit acquisition of a share table lock:

- Your transaction only queries the table and requires a consistent set of the table's data for the duration of the transaction (requires transaction-level read consistency for the locked table).

- It is acceptable if other transactions attempting to update the locked table concurrently must wait until all transactions with the share table locks commit or roll back.
- It is acceptable to allow other transactions to acquire concurrent share table locks on the same table, also allowing them the option of transaction-level read consistency.

---

---

**Caution: Your transaction may or may not update the table later in the same transaction. However, if multiple transactions concurrently hold share table locks for the same table, no transaction can update the table (even if row locks are held as the result of a `SELECT... FOR UPDATE` statement). Therefore, if concurrent share table locks on the same table are common, updates cannot proceed and deadlocks are common. In this case, use share row exclusive or exclusive table locks instead.**

---

---

For example, assume that two tables, `EMP_TAB` and `BUDGET_TAB`, require a consistent set of data in a third table, `DEPT_TAB`. For a given department number, you want to update the information in both of these tables, and ensure that no new members are added to the department between these two transactions.

Although this scenario is quite rare, it can be accommodated by locking the `DEPT_TAB` table in `SHARE MODE`, as shown in the following example. Because the `DEPT_TAB` table is not highly volatile, few, if any, users would need to update it while it was locked for the updates to `EMP_TAB` and `BUDGET_TAB`.

---



---

**Note:** You may need to set up data structures similar to the following for certain examples to work:

```
CREATE TABLE dept_tab(
  deptno NUMBER(2) NOT NULL,
  dname VARCHAR2(14),
  loc VARCHAR2(13));
```

```
CREATE TABLE emp_tab (
  empno NUMBER(4) NOT NULL,
  ename VARCHAR2(10),
  job VARCHAR2(9),
  mgr NUMBER(4),
  hiredate DATE,
  sal NUMBER(7,2),
  comm NUMBER(7,2),
  deptno NUMBER(2));
```

```
CREATE TABLE Budget_tab (
  totalsal NUMBER(7,2),
  deptno NUMBER(2) NOT NULL);
```

---



---

```
LOCK TABLE Dept_tab IN SHARE MODE;
UPDATE Emp_tab
  SET sal = sal * 1.1
  WHERE deptno IN
    (SELECT deptno FROM Dept_tab WHERE loc = 'DALLAS');
UPDATE Budget_tab
  SET Totalsal = Totalsal * 1.1
  WHERE Deptno IN
    (SELECT Deptno FROM Dept_tab WHERE Loc = 'DALLAS');

COMMIT; /* This releases the lock */
```

## SHARE ROW EXCLUSIVE

```
LOCK TABLE Emp_tab IN SHARE ROW EXCLUSIVE MODE;
```

Conditions that warrant the explicit acquisition of a share row exclusive table lock include the following:

- Your transaction requires both transaction-level read consistency for the specified table and the ability to update the locked table.

- You are not concerned about explicit row locks being obtained (via `SELECT... FOR UPDATE`) by other transactions, which may or may not make `UPDATE` and `INSERT` statements in the locking transaction wait to update the table (deadlocks might be observed).
- You only want a single transaction to have the above behavior.

## EXCLUSIVE

```
LOCK TABLE Emp_tab IN EXCLUSIVE MODE;
```

Conditions that warrant the explicit acquisition of an exclusive table lock include the following:

- Your transaction requires immediate update access to the locked table. Therefore, if your transaction holds an exclusive table lock, then other transactions cannot lock specific rows in the locked table.
- Your transaction also observes transaction-level read consistency for the locked table until the transaction is committed or rolled back.
- You are not concerned about low levels of data concurrency, making transactions that request exclusive table locks wait in line to update the table sequentially.

## Privileges Required

You can automatically acquire any type of table lock on tables in your schema; however, to acquire a table lock on a table in another schema, you must have the `LOCK ANY TABLE` system privilege or any object privilege (for example, `SELECT` or `UPDATE`) for the table.

## Letting Oracle Control Table Locking

Letting Oracle control table locking means your application needs less programming logic, but also has less control, than if you manage the table locks yourself.

Issuing the command `SET TRANSACTION ISOLATION LEVEL SERIALIZABLE` or `ALTER SESSION ISOLATION LEVEL SERIALIZABLE` preserves ANSI serializability without changing the underlying locking protocol. This technique allows concurrent access to the table while providing ANSI serializability. Getting table locks greatly reduces concurrency.

Table locks are also controlled by the `ROW_LOCKING` and `SERIALIZABLE` initialization parameters. By default, `SERIALIZABLE` is set to `FALSE` and `ROW_LOCKING` is set to `ALWAYS`. In almost every case, these parameters should not be altered. They are provided for sites that must run in ANSI/ISO compatible mode, or that want to use applications written to run with earlier versions of Oracle. Only these sites should consider altering these parameters, as there is a significant performance degradation caused by using other than the defaults.

**See Also:** *Oracle8i SQL Reference* for details about the `SET TRANSACTION` and `ALTER SESSION` commands.

The settings for these parameters should be changed only when an instance is shut down. If multiple instances are accessing a single database, then all instances should use the same setting for these parameters.

## Summary of Non-Default Locking Options

Three combinations of settings for `SERIALIZABLE` and `ROW_LOCKING`, other than the default settings, are available to change the way locking occurs for transactions. [Table 7-1](#) summarizes the non-default settings and why you might choose to execute your transactions in a non-default way.

**Table 7-1** Summary of Non-Default Locking Options

Case	Description	SERIALIZABLE	ROW_LOCKING
1	Equivalent to Version 5 and earlier Oracle releases (no concurrent inserts, updates, or deletes in a table)	Disabled (default)	INTENT
2	ANSI compatible	Enabled	ALWAYS
3	ANSI compatible, with table-level locking (no concurrent inserts, updates, or deletes in a table)	Enabled	INTENT

[Table 7-2](#) illustrates the difference in locking behavior resulting from the three possible settings of the `SERIALIZABLE` option and `ROW_LOCKING` initialization parameter, as shown in [Table 7-1](#).

**Table 7-2 Non-default Locking Behavior**

STATEMENT	CASE 1		CASE 2		CASE 3	
	row	table	row	table	row	table
SELECT	-	-	-	S	-	S
INSERT	X	SRX	X	RX	X	SRX
UPDATE	X	SRX	X	SRX	X	SRX
DELETE	X	SRX	X	SRX	X	SRX
SELECT...FOR UPDATE	X	RS	X	S	X	S
LOCK TABLE... IN..						
ROW SHARE MODE	-	RS	-	RS	-	RS
ROW EXCLUSIVE MODE	-	RX	-	RX	-	RX
SHARE MODE	-	S	-	S	-	S
SHARE ROW EXCLUSIVE MODE	-	SRX	-	SRX	-	SRX
EXCLUSIVE MODE	-	X	-	X	-	X
DDL statements	-	X	-	X	-	X

## Explicitly Acquiring Row Locks

You can override default locking with a `SELECT` statement that includes the `FOR UPDATE` clause. `SELECT... FOR UPDATE` is used to acquire exclusive row locks for selected rows (as an `UPDATE` statement does) in anticipation of actually updating the selected rows.

You can use a `SELECT... FOR UPDATE` statement to lock a row without actually changing it. For example, several triggers in [Chapter 12, "Using Triggers"](#), show how to implement referential integrity. In the `EMP_DEPT_CHECK` trigger (see "Foreign Key Trigger for Child Table"), the row that contains the referenced parent key value is locked to guarantee that it remains for the duration of the transaction; if the parent key is updated or deleted, referential integrity would be violated.

`SELECT... FOR UPDATE` statements are often used by interactive programs that allow a user to modify fields of one or more specific rows (which might take some time); row locks on the rows are acquired so that only a single interactive program user is updating the rows at any given time.

If a `SELECT... FOR UPDATE` statement is used when defining a cursor, then the rows in the return set are locked before the first fetch, when the cursor is opened; rows

are not individually locked as they are fetched from the cursor. Locks are only released when the transaction that opened the cursor is committed or rolled back; locks are not released when a cursor is closed.

Each row in the return set of a `SELECT... FOR UPDATE` statement is locked individually; the `SELECT... FOR UPDATE` statement waits until the other transaction releases the conflicting row lock. Therefore, if a `SELECT... FOR UPDATE` statement locks many rows in a table, and if the table experiences reasonable update activity, then it would most likely improve performance if you instead acquired an exclusive table lock.

When acquiring row locks with `SELECT... FOR UPDATE`, you can indicate if you do or do not want to wait to acquire the lock. If you specify the `NOWAIT` option, then you only acquire the row lock if it is immediately possible. Otherwise, an error is returned to notify you that the lock is not possible at this time. In this case, you can attempt to lock the row later.

If `NOWAIT` is omitted, then the transaction does not proceed until the requested row lock is acquired. If the wait for a row lock is excessive, then users might want to cancel the lock operation and retry later; you can code such logic into your applications.

As described on "[Choosing a Locking Strategy](#)" on page 7-12, a distributed transaction waiting for a row lock can time-out waiting for the requested lock if the elapsed amount of time reaches the interval set by the initialization parameter `DISTRIBUTED_LOCK_TIMEOUT`.

## User Locks

### Creating User Locks

You can use Oracle Lock Management services for your applications. It is possible to request a lock of a specific mode, give it a unique name recognizable in another procedure in the same or another instance, change the lock mode, and release it. Because a reserved user lock is the same as an Oracle lock, it has all the functionality of an Oracle lock, such as deadlock detection. Be certain that any user locks used in distributed transactions are released upon COMMIT, or an undetected deadlock may occur.

**See Also:** *Oracle8i Supplied PL/SQL Packages Reference* has detailed information on the DBMS\_LOCK package.

### Sample User Locks

Some uses of user locks are:

- Providing exclusive access to a device, such as a terminal
- Providing application-level enforcement of read locks
- Detect when a lock is released and cleanup after the application
- Synchronizing applications and enforce sequential processing

The following Pro\*COBOL precompiler example shows how locks can be used to ensure that there are no conflicts when multiple people need to access a single device.

```
*****
* Print Check *
* Any cashier may issue a refund to a customer returning goods. *
* Refunds under $50 are given in cash, above that by check. *
* This code prints the check. The one printer is opened by all *
* the cashiers to avoid the overhead of opening and closing it *
* for every check. This means that lines of output from multiple*
* cashiers could become interleaved if we don't ensure exclusive*
* access to the printer. The DBMS_LOCK package is used to *
* ensure exclusive access. *
*****
CHECK-PRINT
*
* Get the lock "handle" for the printer lock.
MOVE "CHECKPRINT" TO LOCKNAME-ARR.
```

```

MOVE 10 TO LOCKNAME-LEN.
EXEC SQL EXECUTE
    BEGIN DBMS_LOCK.ALLOCATE_UNIQUE ( :LOCKNAME, :LOCKHANDLE );
    END; END-EXEC.
*
* Lock the printer in exclusive mode (default mode).
EXEC SQL EXECUTE
    BEGIN DBMS_LOCK.REQUEST ( :LOCKHANDLE );
    END; END-EXEC.
* We now have exclusive use of the printer, print the check.

...

*
* Unlock the printer so other people can use it
*
EXEC SQL EXECUTE
    BEGIN DBMS_LOCK.RELEASE ( :LOCKHANDLE );

    END; END-EXEC.

```

## Viewing and Monitoring Locks

Oracle provides two facilities to display locking information for ongoing transactions within an instance:

Enterprise Manager  
Monitors  
(Lock and Latch Monitors)

The Monitor feature of Enterprise Manager provides two monitors for displaying lock information of an instance. Refer to *Oracle Enterprise Manager Administrator's Guide* for complete information about the Enterprise Manager monitors.

UTLLOCKT.SQL

The UTLLOCKT.SQL script displays a simple character lock wait-for graph in tree structured fashion. Using any *ad hoc* SQL tool (such as SQL\*Plus) to execute the script, it prints the sessions in the system that are waiting for locks and the corresponding blocking locks. The location of this script file is operating system dependent. (You must have run the CATBLOCK.SQL script before using UTLLOCKT.SQL.)

## Concurrency Control Using Serializable Transactions

By default, the Oracle Server permits concurrently executing transactions to modify, add, or delete rows in the same table, and in the same data block. Changes made by one transaction are not seen by another concurrent transaction until the transaction that made the changes commits.

If a transaction A attempts to update or delete a row that has been locked by another transaction B (by way of a DML or `SELECT... FOR UPDATE` statement), then A's DML command blocks until B commits or rolls back. Once B commits, transaction A can see changes that B has made to the database.

For most applications, this concurrency model is the appropriate one. In some cases, however, it is advantageous to allow transactions to be serializable. Serializable transactions must execute in such a way that they appear to be executing one at a time (serially), rather than concurrently. In other words, concurrent transactions executing in serialized mode are only permitted to make database changes that they could have made if the transactions were scheduled to run one after the other.

The ANSI/ISO SQL standard SQL92 defines three possible kinds of transaction interaction, and four levels of isolation that provide increasing protection against these interactions. These interactions and isolation levels are summarized in [Table 7-3](#).

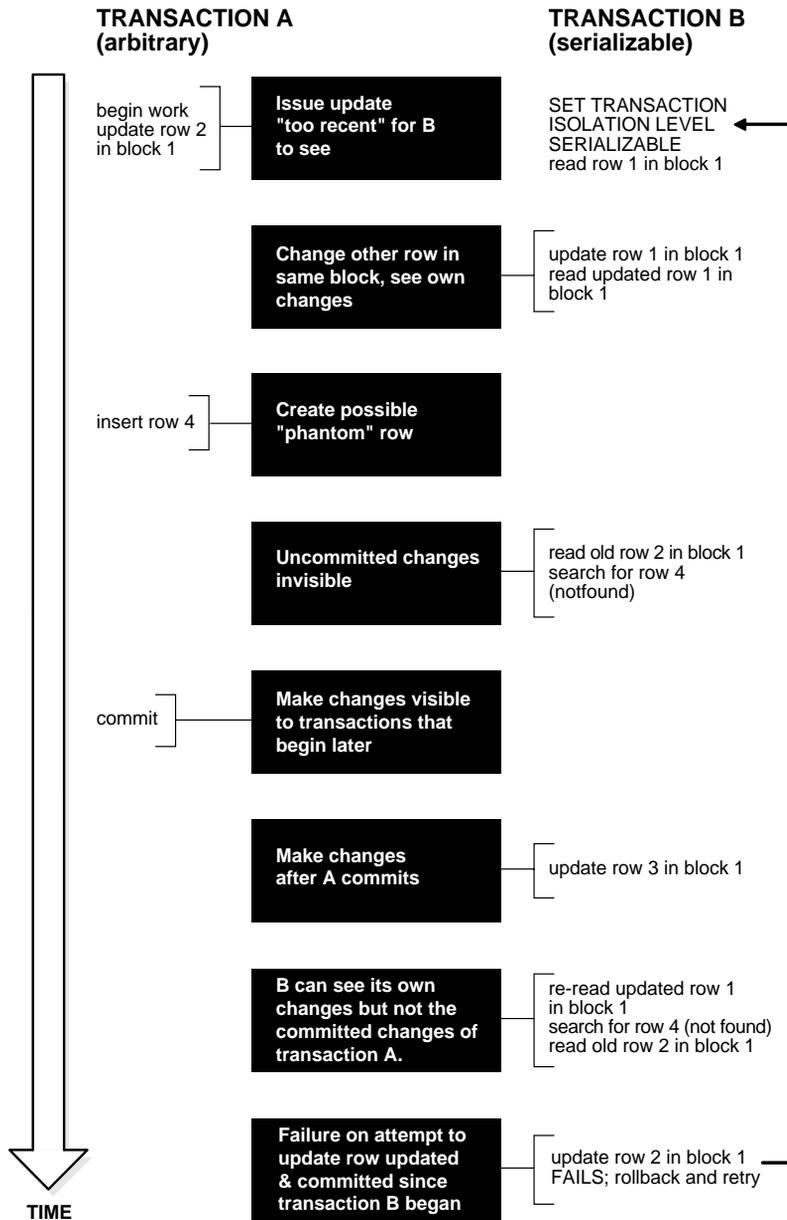
**Table 7-3** *ANSI Isolation Levels*

Isolation Level	Dirty Read (1)	Non-Repeatable Read (2)	Phantom Read (3)
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not possible	Possible	Possible
REPEATABLE READ	Not possible	Not possible	Possible
SERIALIZABLE	Not possible	Not possible	Not possible
Notes:	(1) A transaction can read uncommitted data changed by another transaction. (2) A transaction re-read data committed by another transaction and sees the new data. (3) A transaction can re-execute a query, and discover new rows inserted by another committed transaction.		

The behavior of Oracle with respect to these isolation levels is summarized below:

READ UNCOMMITTED	Oracle never permits "dirty reads." This is not required for high throughput with Oracle.
READ COMMITTED	Oracle meets the READ COMMITTED isolation standard. This is the default mode for all Oracle applications. Note that because an Oracle query only sees data that was committed at the beginning of the query (the snapshot time), Oracle offers more consistency than actually required by the ANSI/ISO SQL92 standards for READ COMMITTED isolation.
REPEATABLE READ	Oracle does not support this isolation level, except as provided by SERIALIZABLE.
SERIALIZABLE	You can set this isolation level using the SET TRANSACTION command or the ALTER SESSION command, as described on.

Figure 7-2 Time Line for Two Transactions



## Serializable Transaction Interaction

**Figure 7-3** on page 7-27 shows how a serializable transaction (Transaction B) interacts with another transaction (A, which can be either `SERIALIZABLE` or `READ COMMITTED`).

When a serializable transaction fails with an `ORA-08177` error ("cannot serialize access"), the application can take any of several actions:

- Commit the work executed to that point
- Execute additional, different, statements, perhaps after rolling back to a prior savepoint in the transaction
- Roll back the entire transaction and try it again

Oracle stores control information in each data block to manage access by concurrent transactions. To use the `SERIALIZABLE` isolation level, you must use the `INITRANS` clause of the `CREATE TABLE` or `ALTER TABLE` command to set aside storage for this control information. To use serializable mode, `INITRANS` must be set to at least 3.

## Setting the Isolation Level

You can change the isolation level of a transaction using the `ISOLATION LEVEL` clause of the `SET TRANSACTION` command. The `SET TRANSACTION` command must be the first command issued in a transaction. If it is not, then the following error is issued:

```
ORA-01453: SET TRANSACTION must be first statement of transaction
```

Use the `ALTER SESSION` command to set the transaction isolation level on a session-wide basis.

**See Also:** *Oracle8i Reference* for the complete syntax of the `SET TRANSACTION` and `ALTER SESSION` commands.

### The `INITRANS` Parameter

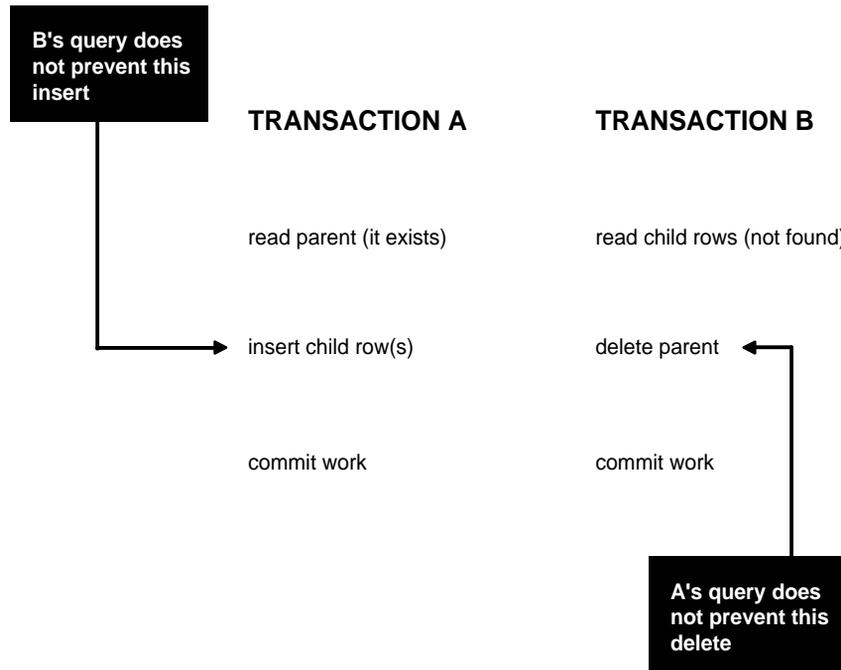
Oracle stores control information in each data block to manage access by concurrent transactions. Therefore, if you set the transaction isolation level to serializable, then you must use the `ALTER TABLE` command to set `INITRANS` to at least 3. This parameter causes Oracle to allocate sufficient storage in each block to record the history of recent transactions that accessed the block. Higher values should be used for tables that will undergo many transactions updating the same blocks.

## Referential Integrity and Serializable Transactions

Because Oracle does not use read locks, even in `SERIALIZABLE` transactions, data read by one transaction can be overwritten by another. Transactions that perform database consistency checks at the application level should not assume that the data they read will not change during the execution of the transaction (even though such changes are not visible to the transaction). Database inconsistencies can result unless such application-level consistency checks are coded carefully, even when using `SERIALIZABLE` transactions. Note, however, that the examples shown in this section are applicable for both `READ COMMITTED` and `SERIALIZABLE` transactions.

**Figure 7-3** on page 7-27 two different transactions that perform application-level checks to maintain the referential integrity parent/child relationship between two tables. One transaction reads the parent table to determine that a row with a specific primary key value exists before inserting corresponding child rows. The other transaction checks to see that no corresponding detail rows exist before proceeding to delete a parent row. In this case, both transactions assume (but do not ensure) that data they read will not change before the transaction completes.

**Figure 7-3 Referential Integrity Check**



Note that the read issued by transaction A does not prevent transaction B from deleting the parent row. Likewise, transaction B's query for child rows does not prevent the insertion of child rows by transaction A. Therefore the above scenario leaves in the database a child row with no corresponding parent row. This result would occur even if both A and B are `SERIALIZABLE` transactions, because neither transaction prevents the other from making changes in the data it reads to check consistency.

As this example illustrates, for some transactions, application developers must specifically ensure that the data read by one transaction is not concurrently written by another. This requires a greater degree of transaction isolation than defined by `SQL92 SERIALIZABLE` mode.

## Using SELECT FOR UPDATE

Fortunately, it is straightforward in Oracle to prevent the anomaly described above. *Transaction A* can use `SELECT FOR UPDATE` to query and lock the parent row and thereby prevent *transaction B* from deleting the row. *Transaction B* can prevent *Transaction A* from gaining access to the parent row by reversing the order of its processing steps. *Transaction B* first deletes the parent row, and then rolls back if its subsequent query detects the presence of corresponding rows in the child table.

Referential integrity can also be enforced in Oracle using database triggers, instead of a separate query as in *Transaction A* above. For example, an `INSERT` into the child table can fire a `PRE-INSERT` row-level trigger to check for the corresponding parent row. The trigger queries the parent table using `SELECT FOR UPDATE`, ensuring that parent row (if it exists) remains in the database for the duration of the transaction inserting the child row. If the corresponding parent row does not exist, then the trigger rejects the insert of the child row.

SQL statements issued by a database trigger execute in the context of the SQL statement that caused the trigger to fire. All SQL statements executed within a trigger see the database in the same state as the triggering statement. Thus, in a `READ COMMITTED` transaction, the SQL statements in a trigger see the database as of the beginning of the triggering statement's execution, and in a transaction executing in `SERIALIZABLE` mode, the SQL statements see the database as of the beginning of the transaction. In either case, the use of `SELECT FOR UPDATE` by the trigger correctly enforces referential integrity, as explained above.

## READ COMMITTED and SERIALIZABLE Isolation

Oracle gives the application developer a choice of two transaction isolation levels with different characteristics. Both the `READ COMMITTED` and `SERIALIZABLE` isolation levels provide a high degree of consistency and concurrency. Both levels provide the contention-reducing benefits of Oracle's "read consistency" multi-version concurrency control model and exclusive row-level locking implementation, and are designed for real-world application deployment. The rest of this section compares the two isolation modes and provides information helpful in choosing between them.

### Transaction Set Consistency

A useful way to describe the `READ COMMITTED` and `SERIALIZABLE` isolation levels in Oracle is to consider the following:

- A collection of database tables (or any set of data)
- A particular sequence of reads of rows in those tables
- The set of transactions committed at any particular time

An operation (a query or a transaction) is "transaction set consistent" if all its reads return data written by the same set of committed transactions. In an operation that is not transaction set consistent, some reads reflect the changes of one set of transactions, and other reads reflect changes made by other transactions. An operation that is not transaction set consistent in effect sees the database in a state that reflects no single set of committed transactions.

Oracle provides transactions executing in `READ COMMITTED` mode with transaction set consistency on a per-statement basis (because all rows read by a query must have been committed before the query began). Similarly, Oracle `SERIALIZABLE` mode provides transaction set consistency on a per-transaction basis, because all statements in a `SERIALIZABLE` transaction execute with respect to an image of the database as of the beginning of the transaction.

In other database systems (unlike in Oracle), a single query run in `READ COMMITTED` mode provides results that are not transaction set consistent. The query is not transaction set consistent, because it may see only a subset of the changes made by another transaction. This means, for example, that a join of a master table with a detail table could see a master record inserted by another transaction, but not the corresponding details inserted by that transaction, or vice versa. Oracle's `READ COMMITTED` mode does not experience this effect, and so provides a greater degree of consistency than read-locking systems.

In read-locking systems, at the cost of preventing concurrent updates, `SQL92 REPEATABLE READ` isolation provides transaction set consistency at the statement level, but not at the transaction level. The absence of phantom protection means two queries issued by the same transaction can see data committed by different sets of other transactions. Only the throughput-limiting and deadlock-susceptible `SERIALIZABLE` mode in these systems provides transaction set consistency at the transaction level.

### Functionality Comparison Summary

[Table 7-4](#) summarizes key similarities and differences between `READ COMMITTED` and `SERIALIZABLE` transactions.

**Table 7-4 Read Committed vs. Serializable Transaction**

	<b>Read Committed</b>	<b>Serializable</b>
Dirty write	Not Possible	Not Possible
Dirty read	Not Possible	Not Possible
Non-repeatable read	Possible	Not Possible
Phantoms	Possible	Not Possible
Compliant with ANSI/ISO SQL 92	Yes	Yes
Read snapshot time	Statement	Transaction
Transaction set consistency	Statement level	Transaction level
Row-level locking	Yes	Yes
Readers block writers	No	No
Writers block readers	No	No
Different-row writers block writers	No	No
Same-row writers block writers	Yes	Yes
Waits for blocking transaction	Yes	Yes
Subject to "can't serialize access" error	No	Yes
Error after blocking transaction aborts	No	No
Error after blocking transaction commits	No	Yes

### Choosing an Isolation Level

Application designers and developers should choose an isolation level that is appropriate to the specific application and workload, and may choose different isolation levels for different transactions. The choice should be based on performance and consistency needs, and consideration of application coding requirements.

For environments with many concurrent users rapidly submitting transactions, designers must assess transaction performance requirements in terms of the expected transaction arrival rate and response time demands, and choose an isolation level that provides the required degree of consistency while satisfying performance expectations. Frequently, for high performance environments, the choice of isolation levels involves making a trade-off between consistency and concurrency (transaction throughput).

Both Oracle isolation modes provide high levels of consistency and concurrency (and performance) through the combination of row-level locking and Oracle's multi-version concurrency control system. Because readers and writers don't block one another in Oracle, while queries still see consistent data, both `READ COMMITTED` and `SERIALIZABLE` isolation provide a high level of concurrency for high performance, without the need for reading uncommitted ("dirty") data.

`READ COMMITTED` isolation can provide considerably more concurrency with a somewhat increased risk of inconsistent results (due to phantoms and non-repeatable reads) for some transactions. The `SERIALIZABLE` isolation level provides somewhat more consistency by protecting against phantoms and non-repeatable reads, and may be important where a read/write transaction executes a query more than once. However, `SERIALIZABLE` mode requires applications to check for the "can't serialize access" error, and can significantly reduce throughput in an environment with many concurrent transactions accessing the same data for update. Application logic that checks database consistency must take into account the fact reads don't block writes in either mode.

## Application Tips

When a transaction runs in serializable mode, any attempt to change data that was changed by another transaction since the beginning of the serializable transaction results in the following error:

```
ORA-08177: Can't serialize access for this transaction.
```

When you get an `ORA-08177` error, the appropriate action is to roll back the current transaction, and re-execute it. After a rollback, the transaction acquires a new transaction snapshot, and the DML operation is likely to succeed.

Because a rollback and repeat of the transaction is required, it is good development practice to put DML statements that might conflict with other concurrent transactions towards the beginning of your transaction, whenever possible.

## Autonomous Transactions

This section gives a brief overview of autonomous transactions and what you can do with them.

---

---

**See Also:** For detailed information on autonomous transactions, see *PL/SQL User's Guide and Reference* and [Chapter 12, "Using Triggers"](#).

---

---

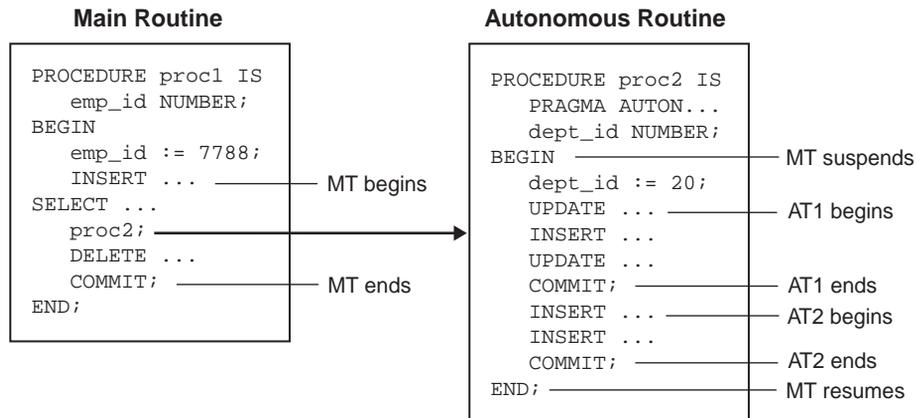
At times, you may want to commit or roll back some changes to a table independently of a primary transaction's final outcome. For example, in a stock purchase transaction, you may want to commit a customer's information regardless of whether the overall stock purchase actually goes through. Or, while running that same transaction, you may want to log error messages to a debug table even if the overall transaction rolls back. Autonomous transactions allow you to do such tasks.

An *autonomous* transaction (AT) is an independent transaction started by another transaction, the *main* transaction (MT). It lets you suspend the main transaction, do SQL operations, commit, or roll back those operations, then resume the main transaction.

An autonomous *transaction* executes within an *autonomous scope*. An *autonomous scope* is a routine you mark with the pragma (compiler directive) `AUTONOMOUS_TRANSACTION`. The pragma instructs the PL/SQL compiler to mark a routine as *autonomous* (independent). In this context, the term *routine* includes:

- Top-level (not nested) anonymous PL/SQL blocks
- Local, stand-alone, and packaged functions and procedures
- Methods of a SQL object type
- PL/SQL triggers

[Figure 7-4](#) shows how control flows from the main routine (MT) to an autonomous routine (AT) and back again. As you can see, the autonomous routine can commit more than one transaction (AT1 and AT2) before control returns to the main routine.

**Figure 7–4 Transaction Control Flow**

When you enter the executable section of an autonomous routine, the main transaction suspends. When you exit the routine, the main transaction resumes. `COMMIT` and `ROLLBACK` end the active autonomous transaction but do not exit the autonomous routine. As [Figure 7–4](#) shows, when one transaction ends, the next SQL statement begins another transaction.

A few more characteristics of autonomous transactions:

- The changes autonomous transactions effect do not depend on the state or the eventual disposition of the main transaction. For example:
  - An autonomous transaction does not see any changes made by the main transaction.
  - When an autonomous transaction commits or rolls back, it does not affect the outcome of the main transaction.
- The changes an autonomous transaction effects are visible to other transactions as soon as that autonomous transaction commits. This means that users can access the updated information without having to wait for the main transaction to commit.
- Autonomous transactions can start other autonomous transactions.

[Figure 7–5](#) illustrates some of the possible sequences autonomous transactions can follow.

**Figure 7-5 Possible Sequences of Autonomous Transactions**

A main transaction scope (MT Scope) begins the main transaction, MTx. MTx invokes the first autonomous transaction scope (AT Scope1). MTx suspends. AT Scope 1 begins the transaction Tx1.1.

At Scope 1 commits or rolls back Tx1.1, then ends. MTx resumes.

MTx invokes AT Scope 2. MT suspends, passing control to AT Scope 2 which, initially, is performing queries.

AT Scope 2 then begins Tx2.1 by, say, doing an update. AT Scope 2 commits or rolls back Tx2.1.

Later, AT Scope 2 begins a second transaction, Tx2.2, then commits or rolls it back.

AT Scope 2 performs a few queries, then ends, passing control back to MTx.

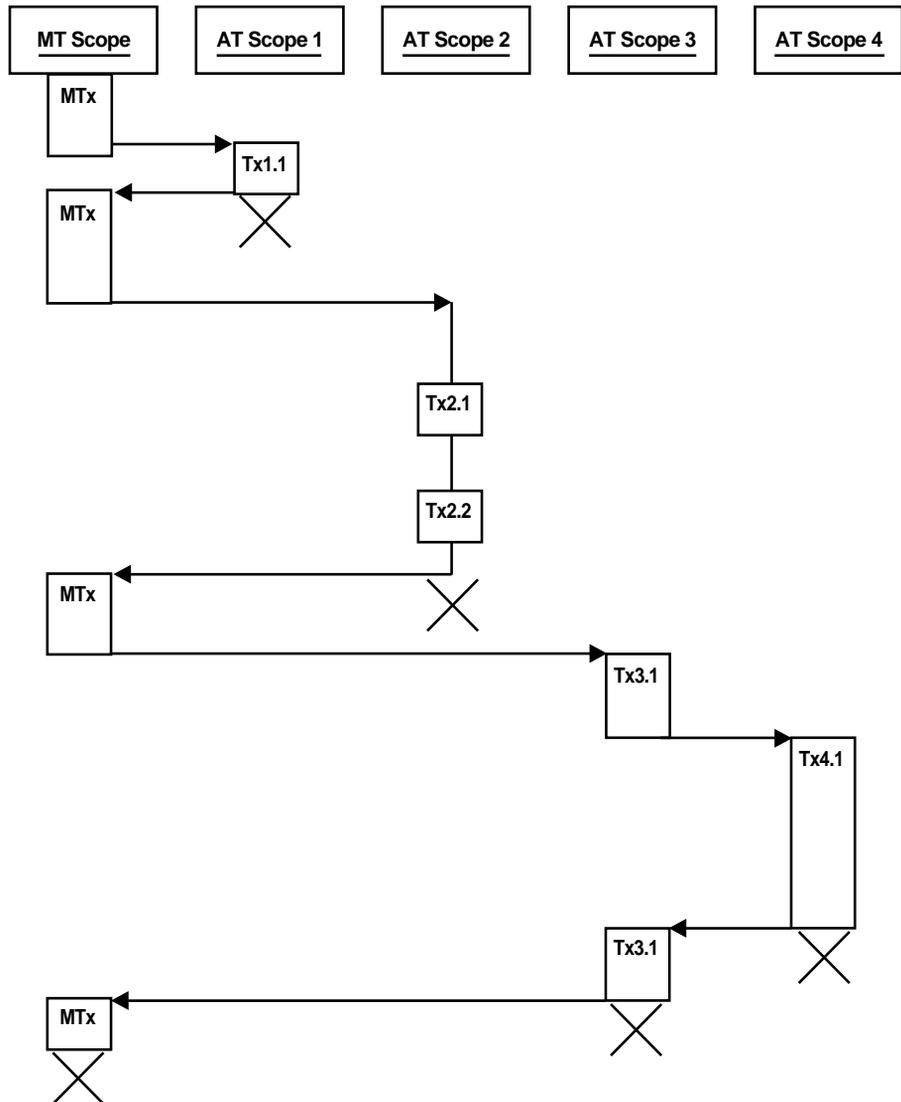
MTx invokes AT Scope 3. MTx suspends, AT Scope 3 begins.

AT Scope 3 begins Tx3.1 which, in turn, invokes AT Scope 4. Tx3.1 suspends, AT Scope 4 begins.

AT Scope 4 begins Tx4.1, commits or rolls it back, then ends. AT Scope 3 resumes.

AT Scope 3 commits or rolls back Tx3.1, then ends. MTx resumes.

Finally, MT Scope commits or rolls back MTx, then ends.



## Examples

The two examples in this section illustrate some of the ways you can use autonomous transactions.

As these examples illustrate, there are four possible outcomes that can occur when you use autonomous and main transactions. The following table presents these possible outcomes. As you can see, there is no dependency between the outcome of an autonomous transaction and that of a main transaction.

<b>Autonomous Transaction</b>	<b>Main Transaction</b>
Commits	Commits
Commits	Rolls back
Rolls back	Commits
Rolls back	Rolls back

### Entering a Buy Order

In this example, a customer enters a buy order. That customer's information (e.g., name, address, phone) is committed to a customer information table—even though the sale does not go through.

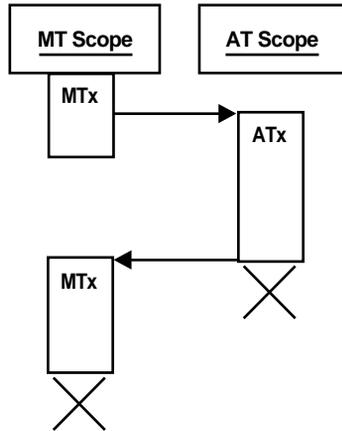
**Figure 7–6 Example: A Buy Order**

MT Scope begins the main transaction, MTx inserts the buy order into a table.

MTx invokes the autonomous transaction scope (AT Scope). When AT Scope begins, MT Scope suspends.

ATx, updates the audit table with customer information.

MTx seeks to validate the order, finds that the selected item is unavailable, and therefore rolls back the main transaction.



### Example: Making a Bank Withdrawal

In the following banking application, a customer tries to make a withdrawal from his or her account. In the process, a main transaction calls one of two autonomous transaction scopes (AT Scope 1, and AT Scope 2).

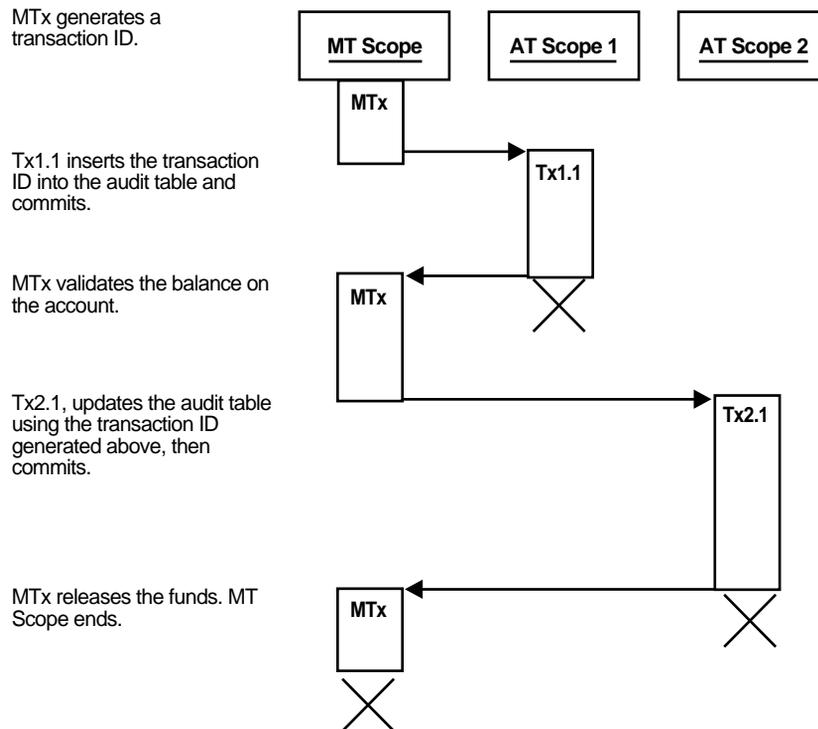
The following diagrams illustrate three possible scenarios for this transaction.

- Scenario 1: There are sufficient funds to cover the withdrawal and therefore the bank releases the funds
- Scenario 2: There are insufficient funds to cover the withdrawal, but the customer has overdraft protection. The bank therefore releases the funds.
- Scenario 3: There are insufficient funds to cover the withdrawal, the customer does not have overdraft protection, and the bank therefore withholds the requested funds.

**Scenario 1:**

There are sufficient funds to cover the withdrawal and therefore the bank releases the funds

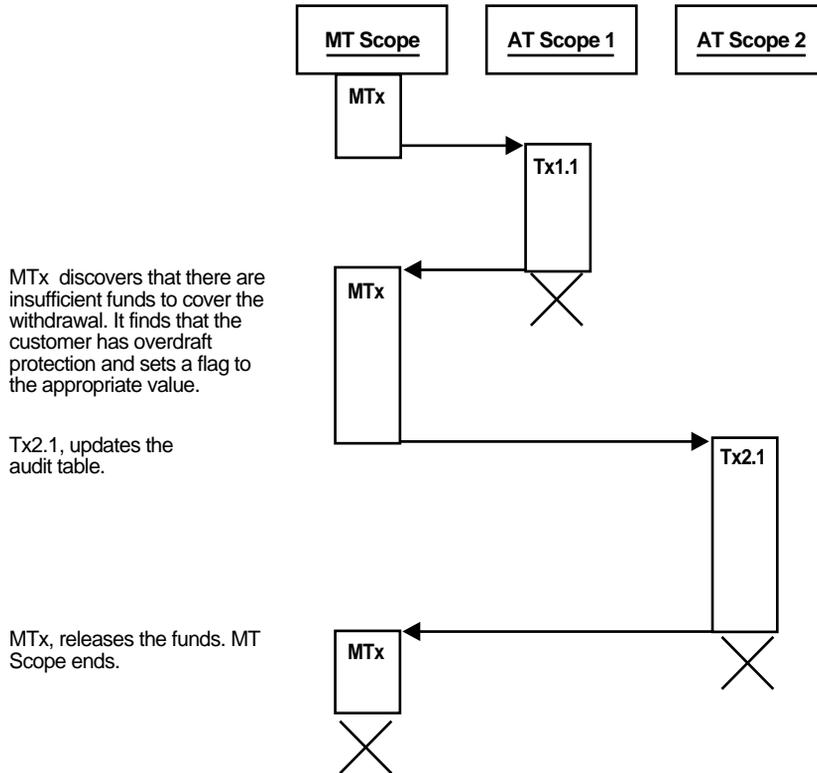
**Figure 7-7 Example: Bank Withdrawal—Sufficient Funds**



**Scenario 2:**

There are insufficient funds to cover the withdrawal, but the customer has overdraft protection. The bank therefore releases the funds.

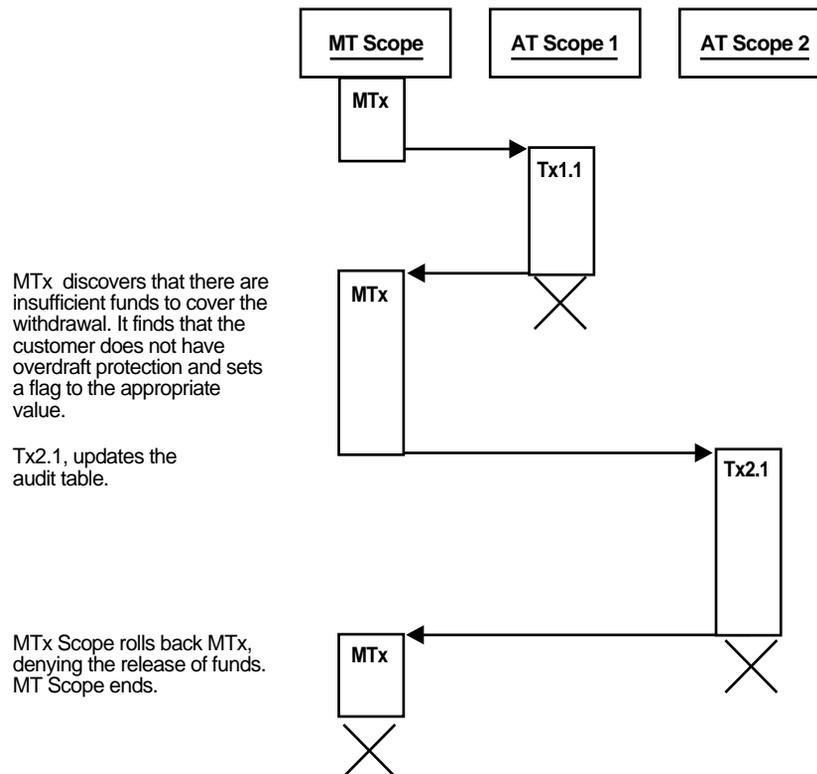
**Figure 7–8 Example: Bank Withdrawal—Insufficient Funds WITH Overdraft Protection**



**Scenario 3:**

There are insufficient funds to cover the withdrawal, the customer does *not* have overdraft protection, and the bank therefore withholds the requested funds.

**Figure 7–9 Example: Bank Withdrawal—Insufficient Funds *WITHOUT* Overdraft Protection**



## Defining Autonomous Transactions

---

---

**Note:** This section is provided here to round out your *general* understanding of autonomous transactions. For a more thorough understanding of autonomous transactions, see *PL/SQL User's Guide and Reference*.

---

---

To define autonomous transactions, you use the pragma (compiler directive) `AUTONOMOUS_TRANSACTION`. The pragma instructs the PL/SQL compiler to mark the procedure, function, or PL/SQL block as autonomous (independent).

You can code the pragma anywhere in the declarative section of a procedure, function, or PL/SQL block. But, for readability, code the pragma at the top of the section. The syntax follows:

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

In the following example, you mark a packaged function as autonomous:

```
CREATE OR REPLACE PACKAGE Banking AS
    FUNCTION Balance (Acct_id INTEGER) RETURN REAL;
    -- add additional functions and/or packages
END Banking;

CREATE OR REPLACE PACKAGE BODY Banking AS
    FUNCTION Balance (Acct_id INTEGER) RETURN REAL IS
        PRAGMA AUTONOMOUS_TRANSACTION;
        My_bal REAL;
    BEGIN
        --add appropriate code
    END;
    -- add additional functions and/or packages...
END Banking;
```

You cannot use the pragma to mark all subprograms in a package (or all methods in an object type) as autonomous. Only individual routines can be marked autonomous. For example, the following pragma is illegal:

```
CREATE OR REPLACE PACKAGE Banking AS
    PRAGMA AUTONOMOUS_TRANSACTION; -- illegal
    FUNCTION Balance (Acct_id INTEGER) RETURN REAL;
END Banking;
```

---

# Dynamic SQL

Dynamic SQL is a programming technique that enables you to build SQL statements dynamically at runtime. You can create more general purpose, flexible applications by using dynamic SQL because the full text of a SQL statement may be unknown at compilation. For example, dynamic SQL lets you create a procedure that operates on a table whose name is not known until runtime.

In past releases of Oracle, the only way to implement dynamic SQL in a PL/SQL application was by using the `DBMS_SQL` package. Oracle8i introduces native dynamic SQL, an alternative to the `DBMS_SQL` package. Using native dynamic SQL, you can place dynamic SQL statements directly into PL/SQL blocks.

This chapter covers the following topics:

- [What Is Dynamic SQL?](#)
- [When to Use Dynamic SQL](#)
- [A Dynamic SQL Scenario Using Native Dynamic SQL](#)
- [Native Dynamic SQL vs. the `DBMS\_SQL` Package](#)
- [Application Development Languages Other Than PL/SQL](#)

## What Is Dynamic SQL?

Dynamic SQL enables you to write programs that reference SQL statements whose full text is not known until runtime. Before discussing dynamic SQL in detail, a clear definition of static SQL may provide a good starting point for understanding dynamic SQL. Static SQL statements do not change from execution to execution. The full text of static SQL statements are known at compilation, which provides the following benefits:

- Successful compilation verifies that the SQL statements reference valid database objects.
- Successful compilation verifies that the necessary privileges are in place to access the database objects.
- Performance of static SQL is generally better than dynamic SQL.

Because of these advantages, you should use dynamic SQL only if you cannot use static SQL to accomplish your goals, or if using static SQL is cumbersome compared to dynamic SQL. However, static SQL has limitations that can be overcome with dynamic SQL. You may not always know the full text of the SQL statements that must be executed in a PL/SQL procedure. Your program may accept user input that defines the SQL statements to execute, or your program may need to complete some processing work to determine the correct course of action. In such cases, you should use dynamic SQL.

For example, consider a reporting application that performs standard queries on tables in a data warehouse environment where the exact table name is unknown until runtime. To accommodate the large amount of data in the data warehouse efficiently, you create a new table every quarter to store the invoice information for the quarter. These tables all have exactly the same definition and are named according to the starting month and year of the quarter, for example `INV_01_1997`, `INV_04_1997`, `INV_07_1997`, `INV_10_1997`, `INV_01_1998`, etc. In such a case, you can use dynamic SQL in your reporting application to specify the table name at runtime.

With static SQL, all of the data definition information, such as table definitions, referenced by the SQL statements in your program must be known at compilation. If the data definition changes, you must change and recompile the program. Dynamic SQL programs can handle changes in data definition information, because the SQL statements can change "on the fly" at runtime. Therefore, dynamic SQL is much more flexible than static SQL. Dynamic SQL enables you to write application code that is reusable because the code defines a process that is independent of the specific SQL statements used.

In addition, dynamic SQL lets you execute SQL statements that are not supported in static SQL programs, such as data definition language (DDL) statements. Support for these statements allows you to accomplish more with your PL/SQL programs.

---

---

**Note:** The phrase *dynamic SQL programs* means programs that include dynamic SQL; such programs also can include static SQL. *Static SQL programs* are those programs that include only static SQL and no dynamic SQL.

---

---

## When to Use Dynamic SQL

You should use dynamic SQL in cases where static SQL does not support the operation you want to perform, or in cases where you do not know the exact SQL statements that must be executed by a PL/SQL procedure. These SQL statements may depend on user input, or they may depend on processing work done by the program. The following sections describe typical situations where you should use dynamic SQL and typical problems that can be solved by using dynamic SQL.

### To Execute Dynamic DML Statements

You can use dynamic SQL to execute DML statements in which the exact SQL statement is not known until runtime. For examples, see the DML examples in the ["Examples of DBMS\\_SQL Package Code and Native Dynamic SQL Code"](#) on page 8-19 and ["Sample DML Operation"](#) on page 8-10.

### To Execute Statements Not Supported by Static SQL in PL/SQL

In PL/SQL, you cannot execute the following types of statements using static SQL:

- Data definition language (DDL) statements, such as `CREATE`, `DROP`, `GRANT`, and `REVOKE`
- Session control language (SCL) statements, such as `ALTER SESSION` and `SET ROLE`

**See Also:** *Oracle8i SQL Reference* for information about DDL and SCL statements.

Use dynamic SQL if you need to execute any of these types of statements within a PL/SQL block.

In addition, static SQL in PL/SQL does not allow the use of the `TABLE` clause in the `SELECT` statements. There is no such limitation in dynamic SQL. For example, the following PL/SQL block contains a `SELECT` statement that uses the `TABLE` clause and native dynamic SQL:

```
CREATE TYPE t_emp AS OBJECT (id NUMBER, name VARCHAR2(20))
/
CREATE TYPE t_emplist AS TABLE OF t_emp
/

CREATE TABLE dept_new (id NUMBER, emps t_emplist)
    NESTED TABLE emps STORE AS emp_table;

INSERT INTO dept_new VALUES (
    10,
    t_emplist(
        t_emp(1, 'SCOTT'),
        t_emp(2, 'BRUCE')));

DECLARE
    deptid NUMBER;
    ename VARCHAR2(20);
BEGIN
    EXECUTE IMMEDIATE 'SELECT d.id, e.name
        FROM dept_new d, TABLE(d.emps) e -- not allowed in static SQL
                                           -- in PL/SQL

        WHERE e.id = 1'
        INTO deptid, ename;
END;
/
```

## To Execute Dynamic Queries

You can use dynamic SQL to create applications that execute dynamic queries, which are queries whose full text is not known until runtime. Many types of applications need to use dynamic queries, including:

- Applications that allow users to input or choose query search or sorting criteria at runtime
- Applications that allow users to input or choose optimizer hints at run time
- Applications that query a database where the data definitions of tables are constantly changing
- Applications that query a database where new tables are created often

For examples, see ["Query Example"](#) on page 8-19, and see the query examples in ["A Dynamic SQL Scenario Using Native Dynamic SQL"](#) on page 8-9.

## To Reference Database Objects that Do Not Exist at Compilation

Many types of applications must interact with data that is generated periodically. For example, it may be possible to determine the definition of the database tables at compilation, but not the names of the tables, because new tables are being generated periodically. Your application needs to access the data, but there is no way to know the exact names of the tables until runtime.

Dynamic SQL can solve this problem, because dynamic SQL allows you to wait until runtime to specify the table names you need to access. For example, in the sample data warehouse application discussed in ["What Is Dynamic SQL?"](#) on page 8-2, new tables are generated every quarter, and these tables always have the same definition. In this case, you might allow a user to specify the name of the table at runtime with a dynamic SQL query similar to the following:

```
CREATE OR REPLACE PROCEDURE query_invoice(
    month VARCHAR2,
    year VARCHAR2) IS
    TYPE cur_typ IS REF CURSOR;
    c cur_typ;
    query_str VARCHAR2(200);
    inv_num NUMBER;
    inv_cust VARCHAR2(20);
    inv_amt NUMBER;
BEGIN
    query_str := 'SELECT num, cust, amt FROM inv_' || month || '_' || year
    || ' WHERE invnum = :id';
    OPEN c FOR query_str USING inv_num;
    LOOP
        FETCH c INTO inv_num, inv_cust, inv_amt;
        EXIT WHEN c%NOTFOUND;
        -- process row here
    END LOOP;
    CLOSE c;
END;
/
```

## To Optimize Execution Dynamically

If you use static SQL, you must decide at compilation how you want to construct your SQL statements, whether to have hints in your statements, and, if you include hints, exactly which hints to have. However, you can use dynamic SQL to build a SQL statement in a way that optimizes the execution and/or concatenates the hints into a SQL statement dynamically. This allows you to change the hints based on your current database statistics, without requiring recompilation.

For example, the following procedure uses a variable called `a_hint` to allow users to pass a hint option to the `SELECT` statement:

```
CREATE OR REPLACE PROCEDURE query_emp
  (a_hint VARCHAR2) AS
  TYPE cur_typ IS REF CURSOR;
  c cur_typ;
BEGIN
  OPEN c FOR 'SELECT ' || a_hint ||
    ' empno, ename, sal, job FROM emp WHERE empno = 7566';
  -- process
END;
/
```

In this example, the user can pass any of the following values for `a_hint`:

- `a_hint = '/**+ ALL_ROWS */'`
- `a_hint = '/**+ FIRST_ROWS */'`
- `a_hint = '/**+ CHOOSE */'`
- Any other valid hint option

**See Also:** *Oracle8i Designing and Tuning for Performance* for more information about using hints.

## To Invoke Dynamic PL/SQL Blocks

You can use the `EXECUTE IMMEDIATE` statement to invoke anonymous PL/SQL blocks. The ability to invoke dynamic PL/SQL blocks can be useful for application extension and customization where the module to be executed is determined dynamically at runtime.

For example, suppose you want to write an application that takes an event number and dispatches to a handler for the event. The name of the handler is in the form `EVENT_HANDLER_event_num`, where `event_num` is the number of the event. One approach would be to implement the dispatcher as a switch statement, as shown below, where the code handles each event by making a static call to its appropriate handler.

```
CREATE OR REPLACE PROCEDURE event_handler_1(param number) AS BEGIN
    -- process event
    RETURN;
END;
/

CREATE OR REPLACE PROCEDURE event_handler_2(param number) AS BEGIN
    -- process event
    RETURN;
END;
/

CREATE OR REPLACE PROCEDURE event_handler_3(param number) AS BEGIN
    -- process event
    RETURN;
END;
/

CREATE OR REPLACE PROCEDURE event_dispatcher
(event number, param number) IS
BEGIN
    IF (event = 1) THEN
        EVENT_HANDLER_1(param);
    ELSIF (event = 2) THEN
        EVENT_HANDLER_2(param);
    ELSIF (event = 3) THEN
        EVENT_HANDLER_3(param);
    END IF;
END;
```

This code is not very extensible because the dispatcher code must be updated whenever a handler for a new event is added. However, using native dynamic SQL, you can write an extensible event dispatcher similar to the following:

```
CREATE OR REPLACE PROCEDURE event_dispatcher
(event NUMBER, param NUMBER) IS
BEGIN
    EXECUTE IMMEDIATE
        'BEGIN
            EVENT_HANDLER_' || to_char(event) || '(:1);
        END; '
    USING param;
END;
/
```

## To Perform Dynamic Operations Using Invoker-Rights

By using the invoker-rights feature with dynamic SQL, you can build applications that issue dynamic SQL statements under the privileges and schema of the invoker. These two features, invoker-rights and dynamic SQL, enable you to build reusable application subcomponents that can operate on and access the invoker's data and modules.

**See Also:** *PL/SQL User's Guide and Reference* for information about using invokers-rights and native dynamic SQL.

## A Dynamic SQL Scenario Using Native Dynamic SQL

The scenario described in this section illustrates the power and flexibility of native dynamic SQL. This scenario includes examples that show you how to perform the following operations using native dynamic SQL:

- Execute DDL and DML operations
- Execute single row and multiple row queries

### Data Model

The database in this scenario is a company's human resources database (named `hr`) with the following data model:

A master table named `offices` contains the list of all company locations. The `offices` table has the following definition:

Column Name	Null?	Type
LOCATION	NOT_NULL	VARCHAR2(200)

Multiple `emp_location` tables contain the employee information, where `location` is the name of city where the office is located. For example, a table named `emp_houston` contains employee information for the company's Houston office, while a table named `emp_boston` contains employee information for the company's Boston office.

Each `emp_location` table has the following definition:

Column Name	Null?	Type
EMPNO	NOT_NULL	NUMBER(4)
ENAME	NOT_NULL	VARCHAR2(10)
JOB	NOT_NULL	VARCHAR2(9)
SAL	NOT_NULL	NUMBER(7,2)
DEPTNO	NOT_NULL	NUMBER(2)

The following sections describe various native dynamic SQL operations that can be performed on the data in the `hr` database.

## Sample DML Operation

The following native dynamic SQL procedure gives a raise to all employees with a particular job title:

```
CREATE OR REPLACE PROCEDURE salary_raise (raise_percent NUMBER, job VARCHAR2) IS
    TYPE loc_array_type IS TABLE OF VARCHAR2(40)
        INDEX BY binary_integer;
    dml_str VARCHAR2(200);
    loc_array loc_array_type;
BEGIN
    -- bulk fetch the list of office locations
    SELECT location BULK COLLECT INTO loc_array
        FROM offices;
    -- for each location, give a raise to employees with the given 'job'
    FOR i IN loc_array.first..loc_array.last LOOP
        dml_str := 'UPDATE emp_' || loc_array(i)
            || ' SET sal = sal * (1+(:raise_percent/100))'
            || ' WHERE job = :job_title';
        EXECUTE IMMEDIATE dml_str USING raise_percent, job;
    END LOOP;
END;
/
```

## Sample DDL Operation

The EXECUTE IMMEDIATE statement can perform DDL operations. For example, the following procedure adds an office location:

```
CREATE OR REPLACE PROCEDURE add_location (loc VARCHAR2) IS
BEGIN
    -- insert new location in master table
    INSERT INTO offices VALUES (loc);
    -- create an employee information table
    EXECUTE IMMEDIATE
    'CREATE TABLE ' || 'emp_' || loc ||
    '(
        empno    NUMBER(4) NOT NULL,
        ename    VARCHAR2(10),
        job      VARCHAR2(9),
        sal      NUMBER(7,2),
        deptno   NUMBER(2)
    )';
END;
/
```

The following procedure deletes an office location:

```
CREATE OR REPLACE PROCEDURE drop_location (loc VARCHAR2) IS
BEGIN
    -- delete the employee table for location 'loc'
    EXECUTE IMMEDIATE 'DROP TABLE ' || 'emp_' || loc;
    -- remove location from master table
    DELETE FROM offices WHERE location = loc;
END;
/
```

## Sample Dynamic Single-Row Query

The EXECUTE IMMEDIATE statement can perform dynamic single-row queries. You can specify bind variables in the USING clause and fetch the resulting row into the target specified in the INTO clause of the statement.

The following function retrieves the number of employees at a particular location performing a specified job:

```
CREATE OR REPLACE FUNCTION get_num_of_employees (loc VARCHAR2, job VARCHAR2)
RETURN NUMBER IS
    query_str VARCHAR2(1000);
    num_of_employees NUMBER;
BEGIN
    query_str := 'SELECT COUNT(*) FROM '
        || ' emp_' || loc
        || ' WHERE job = :job_title';
    EXECUTE IMMEDIATE query_str
        INTO num_of_employees
        USING job;
    RETURN num_of_employees;
END;
/
```

## Sample Dynamic Multiple-Row Query

The OPEN-FOR, FETCH, and CLOSE statements can perform dynamic multiple-row queries. For example, the following procedure lists all of the employees with a particular job at a specified location:

```
CREATE OR REPLACE PROCEDURE list_employees(loc VARCHAR2, job VARCHAR2) IS
    TYPE cur_typ IS REF CURSOR;
    c          cur_typ;
    query_str  VARCHAR2(1000);
    emp_name   VARCHAR2(20);
    emp_num    NUMBER;
BEGIN
    query_str := 'SELECT ename, empno FROM emp_' || loc
                || ' WHERE job = :job_title';
    -- find employees who perform the specified job
    OPEN c FOR query_str USING job;
    LOOP
        FETCH c INTO emp_name, emp_num;
        EXIT WHEN c%NOTFOUND;
        -- process row here
    END LOOP;
    CLOSE c;
END;
/
```

## Native Dynamic SQL vs. the DBMS\_SQL Package

Oracle provides two methods for using dynamic SQL within PL/SQL: native dynamic SQL and the DBMS\_SQL package. Native dynamic SQL enables you to place dynamic SQL statements directly into PL/SQL code. These dynamic statements include DML statements (including queries), PL/SQL anonymous blocks, DDL statements, transaction control statements, and session control statements.

To process most native dynamic SQL statements, you use the EXECUTE IMMEDIATE statement. However, to process a multi-row query (SELECT statement), you use OPEN-FOR, FETCH, and CLOSE statements.

---

---

**Note:** To use native dynamic SQL, the COMPATIBLE initialization parameter must be set to 8.1.0 or higher. See *Oracle8i Migration* for more information about the COMPATIBLE parameter.

---

---

The `DBMS_SQL` package is a PL/SQL library that offers a programmatic API to execute SQL statements dynamically. The `DBMS_SQL` package has programmatic interfaces to open a cursor, parse a cursor, supply binds, etc. Programs that use the `DBMS_SQL` package make calls to this package to perform dynamic SQL operations.

The following sections provide detailed information about the advantages of both methods.

**See Also:** The *PL/SQL User's Guide and Reference* for detailed information about using native dynamic SQL and the *Oracle8i Supplied PL/SQL Packages Reference* for detailed information about using the `DBMS_SQL` package. In the *PL/SQL User's Guide and Reference*, native dynamic SQL is referred to simply as dynamic SQL.

## Advantages of Native Dynamic SQL

Native dynamic SQL provides the following advantages over the `DBMS_SQL` package:

### Ease of Use

Native dynamic SQL is much simpler to use than the `DBMS_SQL` package. Because native dynamic SQL is integrated with SQL, you can use it in the same way that you currently use static SQL within PL/SQL code. In addition, native dynamic SQL code is typically more compact and readable than equivalent code that uses the `DBMS_SQL` package.

The `DBMS_SQL` package is not as easy to use as native dynamic SQL. There are many procedures and functions that must be used in a strict sequence. Typically, performing simple operations requires a large amount of code when you use the `DBMS_SQL` package. You can avoid this complexity by using native dynamic SQL instead.

**Table 8–1** illustrates the difference in the amount of code required to perform the same operation using the DBMS\_SQL package and native dynamic SQL.

**Table 8–1 Code Comparison of DBMS\_SQL Package and Native Dynamic SQL**

DBMS_SQL Package	Native Dynamic SQL
<pre> CREATE PROCEDURE insert_into_table (     table_name  VARCHAR2,     deptnumber  NUMBER,     deptname    VARCHAR2,     location    VARCHAR2) IS cur_hdl        INTEGER; stmt_str       VARCHAR2(200); rows_processed BINARY_INTEGER;  BEGIN     stmt_str := 'INSERT INTO '            table_name    ' VALUES         (:deptno, :dname, :loc)';      -- open cursor     cur_hdl := dbms_sql.open_cursor;      -- parse cursor     dbms_sql.parse(cur_hdl, stmt_str,         dbms_sql.native);      -- supply binds     dbms_sql.bind_variable         (cur_hdl, ':deptno', deptnumber);     dbms_sql.bind_variable         (cur_hdl, ':dname', deptname);     dbms_sql.bind_variable         (cur_hdl, ':loc', location);      -- execute cursor     rows_processed :=     dbms_sql.execute(cur_hdl);      -- close cursor     dbms_sql.close_cursor(cur_hdl);  END; / </pre>	<pre> CREATE PROCEDURE insert_into_table (     table_name  VARCHAR2,     deptnumber  NUMBER,     deptname    VARCHAR2,     location    VARCHAR2) IS     stmt_str    VARCHAR2(200);  BEGIN     stmt_str := 'INSERT INTO '            table_name    ' values         (:deptno, :dname, :loc)';      EXECUTE IMMEDIATE stmt_str         USING             deptnumber, deptname, location;  END; / </pre>

## Performance Improvements

The performance of native dynamic SQL in PL/SQL is comparable to the performance of static SQL because the PL/SQL interpreter has built-in support for native dynamic SQL. Therefore, the performance of programs that use native dynamic SQL is much better than that of programs that use the DBMS\_SQL package. Typically, native dynamic SQL statements perform 1.5 to 3 times better than equivalent statements that use the DBMS\_SQL package. Of course, your performance gains may vary depending on your application.

The DBMS\_SQL package is based on a procedural API and, as a result, incurs high procedure call and data copy overhead. For example, every time you bind a variable, the DBMS\_SQL package copies the PL/SQL bind variable into its space for later use during execution. Similarly, every time you execute a fetch, first the data is copied into the space managed by the DBMS\_SQL package and then the fetched data is copied, one column at a time, into the appropriate PL/SQL variables, resulting in substantial overhead resulting from data copying. In contrast, native dynamic SQL bundles the statement preparation, binding, and execution steps into a single operation, which minimizes the data copying and procedure call overhead and improves performance.

**Performance Tip** When using either native dynamic SQL or the DBMS\_SQL package, you can improve performance by using bind variables, because using bind variables allows Oracle to share a single cursor for multiple SQL statements.

For example, the following native dynamic SQL code does not use bind variables:

```
CREATE OR REPLACE PROCEDURE del_dept (
    my_deptno dept.deptno%TYPE) IS
BEGIN
    EXECUTE IMMEDIATE 'DELETE FROM dept WHERE deptno = ' || to_char (my_deptno);
END;
/
```

For each distinct `my_deptno` variable, a new cursor is created, which can cause resource contention and poor performance. Instead, bind `my_deptno` as a bind variable, as in the following example:

```
CREATE OR REPLACE PROCEDURE del_dept (
    my_deptno dept.deptno%TYPE) IS
BEGIN
    EXECUTE IMMEDIATE 'DELETE FROM dept WHERE deptno = :1' USING my_deptno;
END;
/
```

Here, the same cursor is reused for different values of the bind `my_deptno`, thereby improving performance and scalability.

### Support for User-Defined Types

Native dynamic SQL supports all of the types supported by static SQL in PL/SQL. Therefore, native dynamic SQL provides support for user-defined types, such as user-defined objects, collections, and `REFs`. The `DBMS_SQL` package does not support these user-defined types.

---

---

**Note:** The `DBMS_SQL` package provides limited support for arrays. See the *Oracle8i Supplied PL/SQL Packages Reference* for information.

---

---

### Support for Fetching Into Records

Native dynamic SQL and static SQL both support fetching into records, but the `DBMS_SQL` package does not. With native dynamic SQL, the rows resulting from a query can be directly fetched into PL/SQL records.

In the following example, the rows from a query are fetched into the `emp_rec` record:

```
DECLARE
    TYPE EmpCurTyp IS REF CURSOR;
    c EmpCurTyp;
    emp_rec emp%ROWTYPE;
    stmt_str VARCHAR2(200);
    e_job emp.job%TYPE;

BEGIN
    stmt_str := 'SELECT * FROM emp WHERE job = :1';
    -- in a multi-row query
    OPEN c FOR stmt_str USING 'MANAGER';
    LOOP
        FETCH c INTO emp_rec;
        EXIT WHEN c%NOTFOUND;
    END LOOP;
    CLOSE c;
    -- in a single-row query
    EXECUTE IMMEDIATE stmt_str INTO emp_rec USING 'PRESIDENT';

END;
/
```

## Advantages of the DBMS\_SQL Package

The DBMS\_SQL package provides the following advantages over native dynamic SQL:

### Support for Client-Side Programs

Currently, the DBMS\_SQL package is supported in client-side programs, but native dynamic SQL is not. Every call to the DBMS\_SQL package from the client-side program translates to a PL/SQL remote procedure call (RPC); these calls occur when you need to bind a variable, define a variable, or execute a statement.

### Support for DESCRIBE

The DESCRIBE\_COLUMNS procedure in the DBMS\_SQL package can be used to describe the columns for a cursor opened and parsed through DBMS\_SQL. The functionality is similar to the DESCRIBE command in SQL\*Plus. Native dynamic SQL does not have a DESCRIBE facility.

### Support for Bulk Dynamic SQL

Bulk SQL is the ability to process multiple rows of data in a single DML statement. Bulk SQL improves performance by reducing the amount of context switching between SQL and the host language. Currently, the DBMS\_SQL package supports bulk dynamic SQL.

Although there is no direct support for bulk operations in native dynamic SQL, you can simulate a native dynamic bulk SQL statement by placing the bulk SQL statement in a 'BEGIN ... END' block and executing the block dynamically. This workaround enables you to realize the benefits of bulk SQL within a native dynamic SQL program. For example, the following native dynamic SQL code copies the ename column of one table to another:

```
CREATE TYPE name_array_type IS
  VARRAY(100) OF VARCHAR2(50)
/

CREATE OR REPLACE PROCEDURE copy_ename_column
  (table1 VARCHAR2, table2 VARCHAR2) IS
  ename_col NAME_ARRAY_TYPE;
```

```
BEGIN
  -- bulk fetch the 'ename' column into a VARRAY of VARCHAR2s.
  EXECUTE IMMEDIATE
    'BEGIN
      SELECT ename BULK COLLECT INTO :tab
      FROM ' || table1 || ';
    END; '
  USING OUT ename_col;

  -- bulk insert the 'ename' column into another table.
  EXECUTE IMMEDIATE
    'BEGIN
      FORALL i IN :first .. :last
        INSERT INTO ' || table2 || ' VALUES (:tab(i));
    END; '
  USING ename_col.first, ename_col.last, ename_col;
END;
/
```

### Multiple Row Updates and Deletes with a RETURNING Clause

The DBMS\_SQL package supports statements with a RETURNING clause that update or delete multiple rows. Native dynamic SQL only supports a RETURNING clause if a single row is returned.

**See Also:** ["DML Returning Example"](#) on page 8-22 for examples of DBMS\_SQL package code and native dynamic SQL code that uses a RETURNING clause.

### Support for SQL Statements Larger than 32KB

The DBMS\_SQL package supports SQL statements larger than 32KB; native dynamic SQL does not.

### Reuse of SQL Statements

The PARSE procedure in the DBMS\_SQL package parses a SQL statement once. After the initial parsing, the statement can be used multiple times with different sets of bind arguments.

In contrast, native dynamic SQL prepares a SQL statement for execution each time the statement is used. Statement preparation typically involves parsing, optimization, and plan generation. Preparing a statement each time it is used incurs a small performance penalty. However, Oracle's shared cursor mechanism

minimizes the cost, and the performance penalty is typically trivial when compared to the performance benefits of native dynamic SQL.

## Examples of DBMS\_SQL Package Code and Native Dynamic SQL Code

The following examples illustrate the differences in the code necessary to complete operations with the DBMS\_SQL package and native dynamic SQL. Specifically, the following types of examples are presented:

- A query
- A DML operation
- A DML returning operation

In general, the native dynamic SQL code is more readable and compact, which can improve developer productivity.

### Query Example

The following example includes a dynamic query statement with one bind variable (:jobname) and two select columns (ename and sal):

```
stmt_str := 'SELECT ename, sal FROM emp WHERE job = :jobname';
```

This example queries for employees with the job description SALESMAN in the job column of the emp table. [Table 8-2](#) shows sample code that accomplishes this query using the DBMS\_SQL package and native dynamic SQL.

**Table 8–2 Querying Using the DBMS\_SQL Package and Native Dynamic SQL**

DBMS_SQL Query Operation	Native Dynamic SQL Query Operation
<pre> DECLARE   stmt_str varchar2(200);   cur_hdl int;   rows_processed int;   name varchar2(10);   salary int; BEGIN   cur_hdl := dbms_sql.open_cursor; -- open cursor   stmt_str := 'SELECT ename, sal FROM emp WHERE   job = :jobname';   dbms_sql.parse(cur_hdl, stmt_str, dbms_   sql.native);    -- supply binds (bind by name)   dbms_sql.bind_variable(     cur_hdl, 'jobname', 'SALESMAN');    -- describe defines   dbms_sql.define_column(cur_hdl, 1, name, 200);   dbms_sql.define_column(cur_hdl, 2, salary);    rows_processed := dbms_sql.execute(cur_hdl); --   execute    LOOP     -- fetch a row     IF dbms_sql.fetch_rows(cur_hdl) &gt; 0 then        -- fetch columns from the row       dbms_sql.column_value(cur_hdl, 1, name);       dbms_sql.column_value(cur_hdl, 2, salary);        -- &lt;process data&gt;        ELSE         EXIT;       END IF;   END LOOP;   dbms_sql.close_cursor(cur_hdl); -- close cursor END; / </pre>	<pre> DECLARE   TYPE EmpCurTyp IS REF CURSOR;   cur EmpCurTyp;   stmt_str VARCHAR2(200);   name VARCHAR2(20);   salary NUMBER; BEGIN   stmt_str := 'SELECT ename, sal FROM emp   WHERE job = :1';   OPEN cur FOR stmt_str USING 'SALESMAN';    LOOP     FETCH cur INTO name, salary;     EXIT WHEN cur%NOTFOUND;     -- &lt;process data&gt;   END LOOP;   CLOSE cur; END; / </pre>

## DML Example

The following example includes a dynamic INSERT statement for a table with three columns:

```
stmt_str := 'INSERT INTO dept_new VALUES (:deptno, :dname, :loc)';
```

This example inserts a new row for which the column values are in the PL/SQL variables `deptnumber`, `deptname`, and `location`. [Table 8–3](#) shows sample code that accomplishes this DML operation using the `DBMS_SQL` package and native dynamic SQL.

**Table 8–3 DML Operation Using the DBMS\_SQL Package and Native Dynamic SQL**

DBMS_SQL DML Operation	Native Dynamic SQL DML Operation
<pre> DECLARE   stmt_str VARCHAR2(200);   cur_hdl NUMBER;   deptnumber NUMBER := 99;   deptname VARCHAR2(20);   location VARCHAR2(10);   rows_processed NUMBER; BEGIN   stmt_str := 'INSERT INTO dept_new VALUES (:deptno, :dname, :loc)';   cur_hdl := DBMS_SQL.OPEN_CURSOR;   DBMS_SQL.PARSE(     cur_hdl, stmt_str, DBMS_SQL.NATIVE);   -- supply binds   DBMS_SQL.BIND_VARIABLE     (cur_hdl, ':deptno', deptnumber);   DBMS_SQL.BIND_VARIABLE     (cur_hdl, ':dname', deptname);   DBMS_SQL.BIND_VARIABLE     (cur_hdl, ':loc', location);   rows_processed := dbms_sql.execute(cur_hdl);   -- execute   DBMS_SQL.CLOSE_CURSOR(cur_hdl); -- close END; / </pre>	<pre> DECLARE   stmt_str VARCHAR2(200);   deptnumber NUMBER := 99;   deptname VARCHAR2(20);   location VARCHAR2(10); BEGIN   stmt_str := 'INSERT INTO dept_new VALUES (:deptno, :dname, :loc)';   EXECUTE IMMEDIATE stmt_str     USING deptnumber, deptname, location; END; / </pre>

### DML Returning Example

The following example includes a dynamic `UPDATE` statement that updates the location of a department when given the department number (`deptnumber`) and a new location (`location`), and then returns the name of the department:

```
stmt_str := 'UPDATE dept_new
            SET loc = :newloc
            WHERE deptno = :deptno
            RETURNING dname INTO :dname';
```

This example inserts a new row for which the column values are in the PL/SQL variables `deptnumber`, `deptname`, and `location`. [Table 8-4](#) shows sample code that accomplishes this DML returning operation using the `DBMS_SQL` package and native dynamic SQL.

**Table 8–4 DML Returning Operation Using the DBMS\_SQL Package and Native Dynamic SQL**

DBMS_SQL DML Returning Operation	Native Dynamic SQL DML Returning Operation
<pre> DECLARE   deptname_array dbms_sql.Varchar2_Table;   cur_hdl INT;   stmt_str VARCHAR2(200);   location VARCHAR2(20);   deptnumber NUMBER := 10;   rows_processed NUMBER; BEGIN   stmt_str := 'UPDATE dept_new     SET loc = :newloc     WHERE deptno = :deptno     RETURNING dname INTO :dname';    cur_hdl := dbms_sql.open_cursor;   dbms_sql.parse     (cur_hdl, stmt_str, dbms_sql.native);   -- supply binds   dbms_sql.bind_variable     (cur_hdl, ':newloc', location);   dbms_sql.bind_variable     (cur_hdl, ':deptno', deptnumber);   dbms_sql.bind_array     (cur_hdl, ':dname', deptname_array);   -- execute cursor   rows_processed := dbms_sql.execute(cur_hdl);   -- get RETURNING column into OUT bind array   dbms_sql.variable_value     (cur_hdl, ':dname', deptname_array);   dbms_sql.close_cursor(cur_hdl); END; / </pre>	<pre> DECLARE   deptname_array dbms_sql.Varchar2_Table;   stmt_str VARCHAR2(200);   location VARCHAR2(20);   deptnumber NUMBER := 10;   deptname VARCHAR2(20); BEGIN   stmt_str := 'UPDATE dept_new     SET loc = :newloc     WHERE deptno = :deptno     RETURNING dname INTO :dname';   EXECUTE IMMEDIATE stmt_str     USING location, deptnumber, OUT deptname; END; / </pre>

## Application Development Languages Other Than PL/SQL

So far, the discussion in this chapter has been about PL/SQL support for dynamic SQL. However, you can use other application development languages to implement programs that use dynamic SQL. These application development languages include C/C++, COBOL, and Java.

If you use C/C++, you can develop applications that use dynamic SQL with the Oracle Call Interface (OCI), or you can use the Pro\*C/C++ precompiler to add dynamic SQL extensions to your C code. Similarly, if you use COBOL, you can use the Pro\*COBOL precompiler to add dynamic SQL extensions to your COBOL code. If you use Java, you can develop applications that use dynamic SQL with JDBC.

In the past, the only way to use dynamic SQL in PL/SQL applications was by using the `DBMS_SQL` package. There are a number of limitations to using this package, including performance concerns. Consequently, application developers may have used one of the alternatives to PL/SQL discussed above to implement dynamic SQL. However, with the introduction of native dynamic SQL in PL/SQL, many of the drawbacks to using PL/SQL for dynamic SQL are now eliminated.

If you have an application that uses OCI, Pro\*C/C++, or Pro\*COBOL for dynamic SQL execution, the network roundtrips required to perform dynamic SQL operations may hurt performance. Because these applications typically reside on clients, more network calls are required to complete dynamic SQL operations. If you have this type of application, consider moving the dynamic SQL functionality to stored procedures and stored functions in PL/SQL that use native dynamic SQL. Doing so might improve the performance of your application because the stored procedures can reside on the server, thereby eliminating the network overhead. You can then call the PL/SQL stored procedures and stored functions from the application.

**See Also:** For information about calling Oracle stored procedures and stored functions from non-PL/SQL applications, refer to:

- *Oracle Call Interface Programmer's Guide*
- *Pro\*C/C++ Precompiler Programmer's Guide*
- *Pro\*COBOL Precompiler Programmer's Guide*
- *Oracle8i Java Stored Procedures Developer's Guide*

---

# Using Procedures and Packages

This chapter describes some of the procedural capabilities of Oracle for application development, including:

- [PL/SQL Program Units](#)
- [Wrapping PL/SQL Code](#)
- [Remote Dependencies](#)
- [Cursor Variables](#)
- [Compile-Time Errors](#)
- [Run-Time Error Handling](#)
- [Debugging Stored Procedures](#)
- [Calling Stored Procedures](#)
- [Calling Remote Procedures](#)
- [Calling Stored Functions from SQL Expressions](#)

## PL/SQL Program Units

PL/SQL is a modern, block-structured programming language. It provides several features that make developing powerful database applications very convenient. For example, PL/SQL provides procedural constructs, such as loops and conditional statements, that are not available in standard SQL.

You can directly enter SQL data manipulation language (DML) statements inside PL/SQL blocks, and you can use procedures, supplied by Oracle, to perform data definition language (DDL) statements.

PL/SQL code runs on the server, so using PL/SQL lets you centralize significant parts of your database applications for increased maintainability and security. It also enables you to achieve a significant reduction of network overhead in client/server applications.

---

---

**Note:** Some Oracle tools, such as Oracle Forms, contain a PL/SQL engine that allows you to run PL/SQL locally.

---

---

You can even use PL/SQL for some database applications in place of 3GL programs that use embedded SQL or the Oracle Call Interface (OCI).

PL/SQL program units include:

- [Anonymous Blocks](#)
- [Stored Program Units \(Procedures, Functions, and Packages\)](#)
- [Triggers](#)

**See Also:** For complete information about the PL/SQL language, see the *PL/SQL User's Guide and Reference*.

## Anonymous Blocks

An anonymous block is a PL/SQL program unit that has no name and it does not require the explicit presence of the `BEGIN` and `END` keywords to enclose the executable statements. An anonymous block consists of an optional *declarative* part, an *executable* part, and one or more optional *exception handlers*.

The declarative part declares PL/SQL variables, exceptions, and cursors. The executable part contains PL/SQL code and SQL statements, and can contain nested blocks. Exception handlers contain code that is called when the exception is raised,

either as a predefined PL/SQL exception (such as `NO_DATA_FOUND` or `ZERO_DIVIDE`) or as an exception that you define.

The following short example of a PL/SQL anonymous block prints the names of all employees in department 20 in the `Emp_tab` table, using the `DBMS_OUTPUT` package:

```
DECLARE
    Emp_name    VARCHAR2(10);
    Cursor      c1 IS SELECT Ename FROM Emp_tab
                WHERE Deptno = 20;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO Emp_name;
        EXIT WHEN c1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(Emp_name);
    END LOOP;
END;
```

---

---

**Note:** If you test this block using SQL\*Plus, then enter the statement `SET SERVEROUTPUT ON`, so that output using the `DBMS_OUTPUT` procedures (for example, `PUT_LINE`) is activated. Also, end the example with a slash (`/`) to activate it.

---

---

**See Also:** For complete information about the `DBMS_OUTPUT` package, see *Oracle8i Supplied PL/SQL Packages Reference*.

Exceptions let you handle Oracle error conditions within PL/SQL program logic. This allows your application to prevent the server from issuing an error that could cause the client application to abend. The following anonymous block handles the predefined Oracle exception `NO_DATA_FOUND` (which would result in an `ORA-01403` error if not handled):

```
DECLARE
    Emp_number    INTEGER := 9999;
    Emp_name      VARCHAR2(10);
BEGIN
    SELECT Ename INTO Emp_name FROM Emp_tab
        WHERE Empno = Emp_number; -- no such number
    DBMS_OUTPUT.PUT_LINE('Employee name is ' || Emp_name);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('No such employee: ' || Emp_number);
END;
```

You can also define your own exceptions, declare them in the declaration part of a block, and define them in the exception part of the block. An example follows:

```
DECLARE
    Emp_name      VARCHAR2(10);
    Emp_number    INTEGER;
    Empno_out_of_range EXCEPTION;
BEGIN
    Emp_number := 10001;
    IF Emp_number > 9999 OR Emp_number < 1000 THEN
        RAISE Empno_out_of_range;
    ELSE
        SELECT Ename INTO Emp_name FROM Emp_tab
            WHERE Empno = Emp_number;
        DBMS_OUTPUT.PUT_LINE('Employee name is ' || Emp_name);
    END IF;
EXCEPTION
    WHEN Empno_out_of_range THEN
        DBMS_OUTPUT.PUT_LINE('Employee number ' || Emp_number ||
            ' is out of range.');
```

```
END;
```

**See Also:** ["Run-Time Error Handling"](#) on page 9-43 and see the *PL/SQL User's Guide and Reference*.

Anonymous blocks are usually used interactively from a tool, such as SQL\*Plus, or in a precompiler, OCI, or SQL\*Module application. They are usually used to call stored procedures or to open cursor variables.

**See Also:** ["Cursor Variables"](#) on page 9-38.

## Stored Program Units (Procedures, Functions, and Packages)

A stored procedure, function, or package is a PL/SQL program unit that:

- Has a name.
- Can take parameters, and can return values.
- Is stored in the data dictionary.
- Can be called by many users.

---

---

**Note:** The term *stored procedure* is sometimes used generically for both stored procedures and stored functions. The only difference between procedures and functions is that functions always return a single value to the caller, while procedures do not return a value to the caller.

---

---

### Naming Procedures and Functions

Because a procedure or function is stored in the database, it must be named. This distinguishes it from other stored procedures and makes it possible for applications to call it. Each publicly-visible procedure or function in a schema must have a unique name, and the name must be a legal PL/SQL identifier.

---

---

**Note:** If you plan to call a stored procedure using a stub generated by SQL\*Module, then the stored procedure name must also be a legal identifier in the calling host 3GL language, such as Ada or C.

---

---

### Parameters for Procedures and Functions

Stored procedures and functions can take parameters. The following example shows a stored procedure that is similar to the anonymous block in "[Anonymous Blocks](#)" on page 9-2.

---

---

**Caution:** To execute the following, use CREATE OR REPLACE PROCEDURE...

---

---

```
PROCEDURE Get_emp_names (Dept_num IN NUMBER) IS
    Emp_name      VARCHAR2(10);
    CURSOR        c1 (Depno NUMBER) IS
                  SELECT Ename FROM Emp_tab
                  WHERE deptno = Depno;
BEGIN
    OPEN c1(Dept_num);
    LOOP
        FETCH c1 INTO Emp_name;
        EXIT WHEN c1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(Emp_name);
    END LOOP;
    CLOSE c1;
END;
```

In this stored procedure example, the department number is an input parameter which is used when the parameterized cursor `c1` is opened.

The formal parameters of a procedure have three major parts:

Name	This must be a legal PL/SQL identifier.
Mode	This indicates whether the parameter is an input-only parameter (IN), an output-only parameter (OUT), or is both an input and an output parameter (IN OUT). If the mode is not specified, then IN is assumed.
Datatype	This is a standard PL/SQL datatype.

**Parameter Modes** Parameter modes define the behavior of formal parameters. The three parameter modes, IN (the default), OUT, and IN OUT, can be used with any subprogram. However, avoid using the OUT and IN OUT modes with functions. The purpose of a function is to take no arguments and return a single value. It is poor programming practice to have a function return multiple values. Also, functions should be free from side effects, which change the values of variables not local to the subprogram.

[Table 9–1](#) summarizes the information about parameter modes.

**See Also:** Parameter modes are explained in detail in the *PL/SQL User's Guide and Reference*.

**Table 9–1 Parameter Modes**

IN	OUT	IN OUT
The default.	Must be specified.	Must be specified.
Passes values to a subprogram.	Returns values to the caller.	Passes initial values to a subprogram; returns updated values to the caller.
Formal parameter acts like a constant.	Formal parameter acts like an uninitialized variable.	Formal parameter acts like an initialized variable.
Formal parameter cannot be assigned a value.	Formal parameter cannot be used in an expression; must be assigned a value.	Formal parameter should be assigned a value.
Actual parameter can be a constant, initialized variable, literal, or expression.	Actual parameter must be a variable.	Actual parameter must be a variable.

**Parameter Datatypes** The datatype of a formal parameter consists of one of the following:

- An *unconstrained* type name, such as NUMBER or VARCHAR2.
- A type that is *constrained* using the %TYPE or %ROWTYPE attributes.

---

**Note:** Numerically constrained types such as NUMBER(2) or VARCHAR2(20) are not allowed in a parameter list.

---

### %TYPE and %ROWTYPE Attributes

Use the type attributes %TYPE and %ROWTYPE to constrain the parameter. For example, the Get\_emp\_names procedure specification in "[Parameters for Procedures and Functions](#)" on page 9-5 could be written as the following:

```
PROCEDURE Get_emp_names(Dept_num IN Emp_tab.Deptno%TYPE)
```

This has the Dept\_num parameter take the same datatype as the Deptno column in the Emp\_tab table. The column and table must be available when a declaration using %TYPE (or %ROWTYPE) is elaborated.

Using %TYPE is recommended, because if the type of the column in the table changes, then it is not necessary to change the application code.

If the `Get_emp_names` procedure is part of a package, then you can use previously-declared public (package) variables to constrain a parameter datatype. For example:

```
Dept_number    number(2);
...
PROCEDURE Get_emp_names(Dept_num IN Dept_number%TYPE);
```

Use the `%ROWTYPE` attribute to create a record that contains all the columns of the specified table. The following example defines the `Get_emp_rec` procedure, which returns all the columns of the `Emp_tab` table in a PL/SQL record for the given `empno`:

---

---

**Caution:** To execute the following, use `CREATE OR REPLACE PROCEDURE...`

---

---

```
PROCEDURE Get_emp_rec (Emp_number IN Emp_tab.Empno%TYPE,
                      Emp_ret     OUT Emp_tab%ROWTYPE) IS
BEGIN
    SELECT Empno, Ename, Job, Mgr, Hiredate, Sal, Comm, Deptno
        INTO Emp_ret
        FROM Emp_tab
        WHERE Empno = Emp_number;
END;
```

You could call this procedure from a PL/SQL block as follows:

```
DECLARE
    Emp_row    Emp_tab%ROWTYPE;    -- declare a record matching a
                                -- row in the Emp_tab table
BEGIN
    Get_emp_rec(7499, Emp_row);    -- call for Emp_tab# 7499
    DBMS_OUTPUT.PUT(Emp_row.Ename || ' ' || Emp_row.Empno);
    DBMS_OUTPUT.PUT(' ' || Emp_row.Job || ' ' || Emp_row.Mgr);
    DBMS_OUTPUT.PUT(' ' || Emp_row.Hiredate || ' ' || Emp_row.Sal);
    DBMS_OUTPUT.PUT(' ' || Emp_row.Comm || ' ' || Emp_row.Deptno);
    DBMS_OUTPUT.NEW_LINE;
END;
```

Stored functions can also return values that are declared using `%ROWTYPE`. For example:

```
FUNCTION Get_emp_rec (Dept_num IN Emp_tab.Deptno%TYPE)
    RETURN Emp_tab%ROWTYPE IS ...
```

**Tables and Records** You can pass PL/SQL tables as parameters to stored procedures and functions. You can also pass tables of records as parameters.

---

**Note:** When passing a user defined type, such as a PL/SQL table or record to a remote procedure, to make PL/SQL use the same definition so that the type checker can verify the source, you must create a redundant loop back DBLINK so that when the PL/SQL compiles, both sources 'pull' from the same location.

---

**Default Parameter Values** Parameters can take default values. Use the `DEFAULT` keyword or the assignment operator to give a parameter a default value. For example, the specification for the `Get_emp_names` procedure could be written as the following:

```
PROCEDURE Get_emp_names (Dept_num IN NUMBER DEFAULT 20) IS ...
```

or

```
PROCEDURE Get_emp_names (Dept_num IN NUMBER := 20) IS ...
```

When a parameter takes a default value, it can be omitted from the actual parameter list when you call the procedure. When you do specify the parameter value on the call, it overrides the default value.

---

**Note:** Unlike in an anonymous PL/SQL block, you do not use the keyword `DECLARE` before the declarations of variables, cursors, and exceptions in a stored procedure. In fact, it is an error to use it.

---

## Creating Stored Procedures and Functions

Use a text editor to write the procedure or function. At the beginning of the procedure, place the following statement:

```
CREATE PROCEDURE Procedure_name AS ...
```

For example, to use the example in "[%TYPE and %ROWTYPE Attributes](#)" on page 9-7, create a text (source) file called `get_emp.sql` containing the following code:

```
CREATE PROCEDURE Get_emp_rec (Emp_number IN Emp_tab.Empno%TYPE,
                             Emp_ret OUT Emp_tab%ROWTYPE) AS
BEGIN
```

```
SELECT Empno, Ename, Job, Mgr, Hiredate, Sal, Comm, Deptno
       INTO Emp_ret
       FROM Emp_tab
       WHERE Empno = Emp_number;
END;
/
```

Then, using an interactive tool such as SQL\*Plus, load the text file containing the procedure by entering the following statement:

```
SQLPLUS> @get_emp
```

This loads the procedure into the current schema from the `get_emp.sql` file (`.sql` is the default file extension). Note the slash (/) at the end of the code. This is not part of the code; it just activates the loading of the procedure.

Use the `CREATE [OR REPLACE] FUNCTION...` statement to store functions.

---

---

**Caution:** When developing a new procedure, it is usually much more convenient to use the `CREATE OR REPLACE... PROCEDURE` statement. This replaces any previous version of that procedure in the same schema with the newer version, but note that this is done without warning.

---

---

You can use either the keyword `IS` or `AS` after the procedure parameter list.

**See Also:** *Oracle8i Reference* for the complete syntax of the `CREATE PROCEDURE` and `CREATE FUNCTION` statements.

**Privileges to Create Procedures and Functions** To create a stand-alone procedure or function, or package specification or body, you must meet the following prerequisites:

- You must have the `CREATE PROCEDURE` system privilege to create a procedure or package in your schema, or the `CREATE ANY PROCEDURE` system privilege to create a procedure or package in another user's schema.

---

---

**Note:** To create without errors (to compile the procedure or package successfully) requires the following additional privileges:

- The owner of the procedure or package must be explicitly granted the necessary object privileges for all objects referenced within the body of the code.
  - The owner cannot obtain required privileges through roles.
- 
- 

If the privileges of a procedure's or a package's owner change, then the procedure must be reauthenticated before it is run. If a necessary privilege to a referenced object is revoked from the owner of the procedure or package, then the procedure cannot be run.

The `EXECUTE` privilege on a procedure gives a user the right to run a procedure owned by another user. Privileged users run the procedure under the security domain of the procedure's owner. Therefore, users never need to be granted the privileges to the objects referenced by a procedure. This allows for more disciplined and efficient security strategies with database applications and their users. Furthermore, all procedures and packages are stored in the data dictionary (in the `SYSTEM` tablespace). No quota controls the amount of space available to a user who creates procedures and packages.

---

---

**Note:** Package creation requires a sort. So the user creating the package should be able to create a sort segment in the temporary tablespace with which the user is associated.

---

---

**See Also:** ["Privileges Required to Execute a Procedure"](#) on page 9-52.

## Altering Stored Procedures and Functions

To alter a stored procedure or function, you must first drop it using the `DROP PROCEDURE` or `DROP FUNCTION` statement, then recreate it using the `CREATE PROCEDURE` or `CREATE FUNCTION` statement. Alternatively, use the `CREATE OR REPLACE PROCEDURE` or `CREATE OR REPLACE FUNCTION` statement, which first drops the procedure or function if it exists, then recreates it as specified.

---

---

**Caution:** The procedure or function is dropped without any warning.

---

---

## Dropping Procedures and Functions

A stand-alone procedure, a stand-alone function, a package body, or an entire package can be dropped using the SQL statements `DROP PROCEDURE`, `DROP FUNCTION`, `DROP PACKAGE BODY`, and `DROP PACKAGE`, respectively. A `DROP PACKAGE` statement drops both a package's specification and body.

The following statement drops the `Old_sal_raise` procedure in your schema:

```
DROP PROCEDURE Old_sal_raise;
```

**Privileges to Drop Procedures and Functions** To drop a procedure, function, or package, the procedure or package must be in your schema, or you must have the `DROP ANY PROCEDURE` privilege. An individual procedure within a package cannot be dropped; the containing package specification and body must be re-created without the procedures to be dropped.

## External Procedures

A PL/SQL procedure executing on an Oracle Server can call an external procedure written in a 3GL. The 3GL procedure runs in a separate address space from that of the Oracle Server.

**See Also:** For information about external procedures, see the [Chapter 10, "External Routines"](#).

## PL/SQL Packages

A *package* is an encapsulated collection of related program objects (e.g., procedures, functions, variables, constants, cursors, and exceptions) stored together in the database.

Using packages is an alternative to creating procedures and functions as standalone schema objects. Packages have many advantages over stand-alone procedures and functions. For example, they:

- Let you organize your application development more efficiently.
- Let you grant privileges more efficiently.
- Let you modify package objects without recompiling dependent schema objects.

- Enable Oracle to read multiple package objects into memory at once.
- Can contain global variables and cursors that are available to all procedures and functions in the package.
- Let you *overload* procedures or functions. Overloading a procedure means creating multiple procedures with the same name in the same package, each taking arguments of different number or datatype.

**See Also:** The *PL/SQL User's Guide and Reference* has more information about subprogram name overloading.

The *specification* part of a package *declares* the public types, variables, constants, and subprograms that are visible outside the immediate scope of the package. The *body* of a package *defines* the objects declared in the specification, as well as private objects that are not visible to applications outside the package.

**Example** The following example shows a package specification for a package named `Employee_management`. The package contains one stored function and two stored procedures. The body for this package defines the function and the procedures:

```
CREATE PACKAGE BODY Employee_management AS
    FUNCTION Hire_emp (Name VARCHAR2, Job VARCHAR2,
        Mgr NUMBER, Hiredate DATE, Sal NUMBER, Comm NUMBER,
        Deptno NUMBER) RETURN NUMBER IS
        New_empno    NUMBER(10);

-- This function accepts all arguments for the fields in
-- the employee table except for the employee number.
-- A value for this field is supplied by a sequence.
-- The function returns the sequence number generated
-- by the call to this function.

    BEGIN
        SELECT Emp_sequence.NEXTVAL INTO New_empno FROM dual;
        INSERT INTO Emp_tab VALUES (New_empno, Name, Job, Mgr,
            Hiredate, Sal, Comm, Deptno);
        RETURN (New_empno);
    END Hire_emp;

    PROCEDURE fire_emp(emp_id IN NUMBER) AS

-- This procedure deletes the employee with an employee
-- number that corresponds to the argument Emp_id. If
-- no employee is found, then an exception is raised.

    BEGIN
        DELETE FROM Emp_tab WHERE Empno = Emp_id;
        IF SQL%NOTFOUND THEN
            Raise_application_error(-20011, 'Invalid Employee
                Number: ' || TO_CHAR(Emp_id));
        END IF;
    END fire_emp;

    PROCEDURE Sal_raise (Emp_id IN NUMBER, Sal_incr IN NUMBER) AS

-- This procedure accepts two arguments. Emp_id is a
-- number that corresponds to an employee number.
-- SAL_INCR is the amount by which to increase the
-- employee's salary. If employee exists, then update
-- salary with increase.

    BEGIN
        UPDATE Emp_tab
            SET Sal = Sal + Sal_incr
            WHERE Empno = Emp_id;
```

```
IF SQL%NOTFOUND THEN
    Raise_application_error(-20011, 'Invalid Employee
        Number: ' || TO_CHAR(Emp_id));
END IF;
END Sal_raise;
END Employee_management;
```

---

---

**Note:** If you want to try this example, then first create the sequence number `Emp_sequence`. Do this with the following SQL\*Plus statement:

```
SQL> CREATE SEQUENCE Emp_sequence
> START WITH 8000 INCREMENT BY 10;
```

---

---

## PL/SQL Object Size Limitation

The size limitation for PL/SQL stored database objects such as procedures, functions, triggers, and packages is the size of the DIANA in the shared pool in bytes. The UNIX limit on the size of the flattened DIANA/pcode size is 64K but the limit may be 32K on desktop platforms such as DOS and Windows.

The most closely related number that a user can access is the `PARSED_SIZE` in the data dictionary view `USER_OBJECT_SIZE`. That gives the size of the DIANA in bytes as stored in the `SYS.IDL_xxx$` tables. This is not the size in the shared pool. The size of the DIANA part of PL/SQL code (used during compilation) is significantly larger in the shared pool than it is in the system table.

**Size Limitation by Version** The size limitation of a PL/SQL package is approximately 128K parsed size in release 7.3. For releases earlier than 7.3 the limitation is 64K.

## Creating Packages

Each part of a package is created with a different statement. Create the package specification using the `CREATE PACKAGE` statement. The `CREATE PACKAGE` statement declares public package objects.

To create a package body, use the `CREATE PACKAGE BODY` statement. The `CREATE PACKAGE BODY` statement defines the procedural code of the public procedures and functions declared in the package specification.

You can also define private, or local, package procedures, functions, and variables in a package body. These objects can only be accessed by other procedures and functions in the body of the same package. They are not visible to external users, regardless of the privileges they hold.

It is often more convenient to add the `OR REPLACE` clause in the `CREATE PACKAGE` or `CREATE PACKAGE BODY` statements when you are first developing your application. The effect of this option is to drop the package or the package body without warning. The `CREATE` statements would then be the following:

```
CREATE OR REPLACE PACKAGE Package_name AS ...
```

and

```
CREATE OR REPLACE PACKAGE BODY Package_name AS ...
```

**Creating Packaged Objects** The body of a package can contain include:

- Procedures and functions declared in the package specification.
- Definitions of cursors declared in the package specification.
- Local procedures and functions, not declared in the package specification.
- Local variables.

Procedures, functions, cursors, and variables that are declared in the package specification are *global*. They can be called, or used, by external users that have `EXECUTE` permission for the package or that have `EXECUTE ANY PROCEDURE` privileges.

When you create the package body, make sure that each procedure that you define in the body has the same parameters, *by name, datatype, and mode*, as the declaration in the package specification. For functions in the package body, the parameters *and* the return type must agree in name and type.

**Privileges to Create or Drop Packages** The privileges required to create or drop a package specification or package body are the same as those required to create or drop a stand-alone procedure or function.

**See Also:** ["Privileges to Create Procedures and Functions"](#) on page 9-10 and ["Privileges to Drop Procedures and Functions"](#) on page 9-12.

### **Naming Packages and Package Objects**

The names of a package and all public objects in the package must be unique within a given schema. The package specification and its body must have the same name. All package constructs must have unique names within the scope of the package, unless overloading of procedure names is desired.

## Package Invalidations and Session State

Each session that references a package object has its own instance of the corresponding package, including persistent state for any public and private variables, cursors, and constants. If any of the session's instantiated packages (specification or body) are subsequently invalidated and recompiled, then all other dependent package instantiations (including state) for the session are lost.

For example, assume that session *S* instantiates packages *P1* and *P2*, and that a procedure in package *P1* calls a procedure in package *P2*. If *P1* is invalidated and recompiled (for example, as the result of a DDL operation), then the session *S* instantiations of both *P1* and *P2* are lost. In such situations, a session receives the following error the first time it attempts to use any object of an invalidated package instantiation:

```
ORA-04068: existing state of packages has been discarded
```

The second time a session makes such a package call, the package is reinstantiated for the session without error.

---

---

**Note:** Oracle has been optimized to not return this message to the session calling the package that it invalidated. Thus, in the example above, session *S* receives this message the first time it called package *P2*, but it does not receive it when calling *P1*.

---

---

In most production environments, DDL operations that can cause invalidations are usually performed during inactive working hours; therefore, this situation might not be a problem for end-user applications. However, if package specification or body invalidations are common in your system during working hours, then you might want to code your applications to detect for this error when package calls are made.

## Oracle Supplied Packages

There are many built-in packages provided with the Oracle Server, either to extend the functionality of the database or to give PL/SQL access to SQL features. You may take advantage of the functionality provided by these packages when creating your application, or you may simply want to use these packages for ideas in creating your own stored procedures.

This section lists each of the supplied packages and indicates where they are described in more detail. These packages run as the calling user, rather than the

package owner. Unless otherwise noted, the packages are callable through public synonyms of the same name.

**Table 9–2 List of Oracle Supplied Packages**

<b>Package Name</b>	<b>Description</b>	<b>Documentation</b>
Calendar (see Note Nr. 2 below)	Provides calendar maintenance functions.	<i>Oracle8i Time Series User's Guide</i>
DBMS_ALERT	Provides support for the asynchronous notification of database events.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_APPLICATION_INFO	Lets you register an application name with the database for auditing or performance tracking purposes.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_AQ	Lets you add a message (of a predefined object type) onto a queue or to dequeue a message.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_AQADM	Lets you perform administrative functions on a queue or queue table for messages of a predefined object type.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_BACKUP_RESTORE (Windows NT only)	Normalizes filenames for Windows NT environments.	<i>Oracle8i Migration</i>
DBMS_DDL	Provides access to some SQL DDL statements from stored procedures, and provides special administration operations not available as DDLs.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_DEBUG	A PL/SQL API to the PL/SQL debugger layer, Probe, in the Oracle server.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_DEFER	Provides the user interface to a replicated transactional deferred remote procedure call facility. Requires the Distributed Option.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_DEFER_QUERY	Permits querying the deferred remote procedure calls (RPC) queue data that is not exposed through views. Requires the Distributed Option.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_DEFER_SYS	Provides the system administrator interface to a replicated transactional deferred remote procedure call facility. Requires the Distributed Option.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_DESCRIBE	Describes the arguments of a stored procedure with full name translation and security checking.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>

**Table 9–2 List of Oracle Supplied Packages**

<b>Package Name</b>	<b>Description</b>	<b>Documentation</b>
DBMS_DISTRIBUTED_TRUST_ADMIN	Maintains the Trusted Database List, which is used to determine if a privileged database link from a particular server can be accepted.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_HS	Lets you create and modify objects in the Heterogeneous Services dictionary.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_HS_PASSTHROUGH	Lets you use Heterogeneous Services to send pass-through SQL statements to non-Oracle systems.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_IOT	Creates a table into which references to the chained rows for an Index Organized Table can be placed using the ANALYZE command.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_JOB	Lets you schedule administrative procedures that you want performed at periodic intervals; it is also the interface for the job queue.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_LOB	Provides general purpose routines for operations on Oracle Large Object (LOBs) datatypes - BLOB, CLOB (read-write), and BFILES (read-only).	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_LOCK	Lets you request, convert and release locks through Oracle Lock Management services.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_LOGMNR	Provides functions to initialize and run the log reader.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_MVIEW	Synonym for DBMS_SNAPSHOT. Lets you refresh snapshots that are not part of the same refresh group and purge logs. Requires the Distributed Option.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_OBFUSCATION_TOOLKIT	Provides Data Encryption Standard procedures.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_OFFLINE_OG	Provides public APIs for offline instantiation of master groups.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_OFFLINE_SNAPSHOT	Provides public APIs for offline instantiation of snapshots.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_OLAP	Provides procedures for summaries, dimensions, and query rewrites.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_ORACLE_TRACE_AGENT	Provides client callable interfaces to the Oracle TRACE instrumentation within the Oracle7 Server.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>

**Table 9–2 List of Oracle Supplied Packages**

<b>Package Name</b>	<b>Description</b>	<b>Documentation</b>
DBMS_ORACLE_TRACE_USER	Provides public access to the Oracle release 7 Server Oracle TRACE instrumentation for the calling user.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_OUTPUT	Accumulates information in a buffer so that it can be retrieved out later.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_PCLXUTIL	Provides intra-partition parallelism for creating partition-wise local indexes.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_PIPE	Provides a DBMS pipe service which enables messages to be sent between sessions.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_PROFILER	Provides a Probe Profiler API to profile existing PL/SQL applications and identify performance bottlenecks.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_RANDOM	Provides a built-in random number generator.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_RECTIFIER_DIFF	Provides APIs used to detect and resolve data inconsistencies between two replicated sites.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_REFRESH	Lets you create groups of snapshots that can be refreshed together to a transactionally consistent point in time. Requires the Distributed Option.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_REPAIR	Provides data corruption repair procedures.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_REPCAT	Provides routines to administer and update the replication catalog and environment. Requires the Replication Option.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_REPCAT_ADMIN	Lets you create users with the privileges needed by the symmetric replication facility. Requires the Replication Option.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_REPCAT_INSTANTIATE	Instantiates deployment templates. Requires the Replication Option.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_REPCAT_RGT	Controls the maintenance and definition of refresh group templates. Requires the Replication Option.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_REPUTIL	Provides routines to generate shadow tables, triggers, and packages for table replication.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>

**Table 9–2 List of Oracle Supplied Packages**

<b>Package Name</b>	<b>Description</b>	<b>Documentation</b>
DBMS_RESOURCE_MANAGER	Maintains plans, consumer groups, and plan directives; it also provides semantics so that you may group together changes to the plan schema.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_RESOURCE_MANAGER_PRIVS	Maintains privileges associated with resource consumer groups.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_RLS	Provides row level security administrative interface.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_ROWID	Provides procedures to create ROWIDs and to interpret their contents.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_SESSION	Provides access to SQL ALTER SESSION statements, and other session information, from stored procedures.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_SHARED_POOL	Lets you keep objects in shared memory, so that they will not be aged out with the normal LRU mechanism.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_SNAPSHOT (synonym DBMS_MVIEW)	Lets you refresh snapshots that are not part of the same refresh group and purge logs. Requires the Distributed Option.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_SPACE	Provides segment space information not available through standard SQL.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_SPACE_ADMIN	Provides tablespace and segment space administration not available through the standard SQL.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_SQL	Lets you use dynamic SQL to access the database.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_STANDARD (See Note Nr. 1 below)	Provides language facilities that help your application interact with Oracle.	Not documented.
DBMS_STATS	Provides a mechanism for users to view and modify optimizer statistics gathered for database objects.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_TRACE	Provides routines to start and stop PL/SQL tracing.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_TRANSACTION	Provides access to SQL transaction statements from stored procedures and monitors transaction activities.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>

**Table 9–2 List of Oracle Supplied Packages**

<b>Package Name</b>	<b>Description</b>	<b>Documentation</b>
DBMS_TTS	Checks if the transportable set is self-contained.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DBMS_UTILITY	Provides various utility routines.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
DEBUG_EXTPROC	Lets you debug external procedures on platforms with debuggers that can attach to a running process.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
OUTLN_PKG	Provides the interface for procedures and functions associated with management of stored outlines.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
PLITBLM (see Note Nr. 1 below)	Handles index-table operations.	Not documented.
SDO_ADMIN (see Note Nr. 3 below)	Provides functions implementing spatial index creation and maintenance for spatial objects.	<i>Oracle Spatial User's Guide and Reference</i>
SDO_GEOM (see Note Nr. 3 below)	Provides functions implementing geometric operations on spatial objects.	<i>Oracle Spatial User's Guide and Reference</i>
SDO_MIGRATE (see Note Nr. 3 below)	Provides functions for migrating spatial data from release 7.3.3 and 7.3.4 to 8.1.x.	<i>Oracle Spatial User's Guide and Reference</i>
SDO_TUNE (see Note Nr. 3 below)	Provides functions for selecting parameters that determine the behavior of the spatial indexing scheme used in the Spatial Cartridge.	<i>Oracle Spatial User's Guide and Reference</i>
STANDARD (see Note Nr. 1 below)	Declares types, exceptions, and subprograms which are available automatically to every PL/SQL program.	Not documented.
TimeSeries (see Note Nr. 2 below)	Provides functions that perform operations, such as extraction, retrieval, arithmetic, and aggregation, on time series data.	<i>Oracle8i Time Series User's Guide</i>
TimeScale (see Note Nr. 2 below)	Provides scaleup and scaledown functions.	<i>Oracle8i Time Series User's Guide</i>
TSTools (see Note Nr. 2 below)	Provides administrative tools procedures.	<i>Oracle8i Time Series User's Guide</i>
UTIL_COLL	Enables PL/SQL programs to use collection locators to query and update.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>

**Table 9–2 List of Oracle Supplied Packages**

Package Name	Description	Documentation
UTL_FILE	Enables your PL/SQL programs to read and write operating system (OS) text files and provides a restricted version of standard OS stream file I/O.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
UTL_HTTP	Enables HTTP callouts from PL/SQL and SQL to access data on the Internet or to call Oracle Web Server Cartridges.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
UTL_INADDR	Provides internet addressing.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
UTL_PG	Provides functions for converting COBOL numeric data into Oracle numbers and Oracle numbers into COBOL numeric data.	<i>Oracle Procedural Gateway for APPC User's Guide</i>
UTL_RAW	Provides SQL functions for RAW datatypes that concat, substr, etc. to and from RAWs .	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
UTL_REF	Enables a PL/SQL program to access an object by providing a reference to the object.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
UTL_SMTP	Provides PL/SQL functionality to send emails.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
UTL_TCP	Provides PL/SQL functionality to support simple TCP/IP-based communications between servers and the outside world.	<i>Oracle8i Supplied PL/SQL Packages Reference</i>
Vir_Pkg (see Note Nr. 2 below)	Provides analytical and conversion functions for Visual Information Retrieval.	<i>Oracle8i Visual Information Retrieval User's Guide and Reference</i>

**Note Nr. 1:**

The DBMS\_STANDARD, STANDARD, and PLITBLM packages contain subprograms to help implement basic language features. Oracle does not recommend that the subprograms be directly called. For this reason, these supplied packages are not documented.

**Note Nr. 2:**

Time-Series, Image, Visual Information Retrieval, Audio, and Server-Managed Video Cartridge packages are installed in user ORDSYS without public synonyms.

**Note Nr. 3:**

Spatial Cartridge packages are installed in user MDSYS with public synonyms.

## Bulk Binds

Oracle uses two engines to run PL/SQL blocks and subprograms: the PL/SQL engine and the SQL engine. The PL/SQL engine runs procedural statements, while the SQL engine runs SQL statements. During execution, every SQL statement causes a context switch between the two engines, which results in a performance penalty.

Performance can be improved substantially by minimizing the number of context switches required to run a particular block or subprogram. When a SQL statement runs inside a loop that uses collection elements as bind variables, the large number of context switches required by the block can cause poor performance. Collections include the following:

- Varrays
- Nested tables
- Index-by tables
- Host arrays

*Binding* is the assignment of values to PL/SQL variables in SQL statements. *Bulk binding* is binding an entire collection at once. Without bulk binds, the elements in a collection are sent to the SQL engine individually, whereas bulk binds pass the entire collection back and forth between the two engines.

Using bulk binds, you can improve performance by reducing the number of context switches required to run SQL statements that use collection elements. Typically, using bulk binds improves performance for SQL statements that affect four or more database rows. The more rows affected by a SQL statement, the greater the performance gain will be with bulk binds.

---

---

**Note:** This section provides an overview of bulk binds to help you decide if you should use them in your PL/SQL applications. For detailed information about using bulk binds, see the *PL/SQL User's Guide and Reference*.

---

---

---



---

**Caution:** You may need to set up or drop data structures for certain examples to work.

---



---

**When to Use Bulk Binds** The following sections discuss common scenarios where bulk binds can improve performance. If you have, or plan to have, similar scenarios in your applications, then you should consider using bulk binds.

**DML Statements Referencing Collections** Bulk binds can be used to improve the performance of DML statements that reference collections. To bulk-bind an input collection before sending it to the SQL engine, use the `FORALL` keyword. The SQL statement must be an `INSERT`, `UPDATE`, or `DELETE` statement that references collection elements.

For example, the following PL/SQL block increases the salary for employees whose manager's ID number is 7902, 7698, or 7839, without using bulk binds:

```
DECLARE
    TYPE Numlist IS VARRAY (100) OF NUMBER;
    Id NUMLIST := NUMLIST(7902, 7698, 7839);
BEGIN
    FOR i IN Id.FIRST..Id.LAST LOOP
        UPDATE Emp_tab SET Sal = 1.1 * Sal
            WHERE Mgr = Id(i);
    END LOOP;
END;
```

To run this block, PL/SQL sends a SQL statement to the SQL engine for each employee that is updated. If there are many employees to update, then the large number of context switches between the PL/SQL engine and the SQL engine can hurt performance.

Use the `FORALL` keyword to bulk-bind the collection and improve performance:

```
DECLARE
    TYPE Numlist IS VARRAY (100) OF NUMBER;
    Id NUMLIST := NUMLIST(7902, 7698, 7839);
BEGIN
    FORALL i IN Id.FIRST..Id.LAST -- bulk-bind the VARRAY
        UPDATE Emp_tab SET Sal = 1.1 * Sal
            WHERE Mgr = Id(i);
END;
```

**SELECT Statements Referencing Collections** Bulk binds can be used to improve the performance of `SELECT` statements that reference collections. To bulk-bind output collections before returning them to the PL/SQL engine, use the keywords `BULK COLLECT INTO`.

For example, the following PL/SQL block returns the employee name and job for employees whose manager's ID number is 7698, without using bulk binds:

```
DECLARE
    TYPE Var_tab IS TABLE OF VARCHAR2(20) INDEX BY BINARY_INTEGER;
    Empno VAR_TAB;
    Ename VAR_TAB;
    Counter NUMBER;
    CURSOR C IS
        SELECT Empno, Ename FROM Emp_tab WHERE Mgr = 7698;
BEGIN

-- Initialize variable tracing number of employees returned.

    counter := 1;

-- Find all employees whose manager's ID number is 7698.

    FOR rec IN C LOOP
        Empno(counter) := rec.Empno;
        Ename(counter) := rec.Ename;
        Counter := Counter + 1;
    END LOOP;
END;
```

PL/SQL sends a SQL statement to the SQL engine for each employee that is selected. If there are many employees selected, then the large number of context switches between the PL/SQL engine and the SQL engine can hurt performance.

Use the `BULK COLLECT INTO` keywords to bulk-bind the collection and improve performance:

```
DECLARE
    TYPE Emplist IS VARRAY(100) OF NUMBER;
    Empids EMPLIST := EMPLIST(7369, 7499, 7521, 7566, 7654, 7698);
    TYPE Bonlist IS TABLE OF Emp_tab.Sal%TYPE;
    Bonlist_inst BONLIST;
BEGIN
    Bonlist_inst := BONLIST(1,2,3,4,5);
    FORALL i IN Empids.FIRST..empIDs.LAST
        UPDATE Emp_tab SET Bonus = 0.1 * Sal
```

```

        WHERE empno = Empids(i)
        RETURNING Sal BULK COLLECT INTO Bonlist_inst;
END;
```

**FOR Loops that Reference Collections and the Returning Into Clause** Bulk binds can be used to improve the performance of FOR loops that reference collections and return DML. If you have, or plan to have, PL/SQL code that does this, then you can use the FORALL keyword along with the BULK COLLECT INTO keywords to improve performance.

For example, the following PL/SQL block updates the Emp\_tab table by computing bonuses for a collection of employees; then it returns the bonuses in a column called Bonlist. Both actions are performed without using bulk binds:

```

DECLARE
    TYPE Emplist IS VARRAY(100) OF NUMBER;
    Empids EMPLIST := EMPLIST(7369, 7499, 7521, 7566, 7654, 7698);
    TYPE Bonlist IS TABLE OF Emp_tab.sal%TYPE;
    Bonlist_inst BONLIST;
BEGIN
    Bonlist_inst := BONLIST(1,2,3,4,5);
    FOR i IN Empids.FIRST..Empids.LAST LOOP
        UPDATE Emp_tab Set Bonus = 0.1 * sal
            WHERE Empno = Empids(i)
            RETURNING Sal INTO BONLIST(i);
    END LOOP;
END;
```

PL/SQL sends a SQL statement to the SQL engine for each employee that is updated. If there are many employees updated, then the large number of context switches between the PL/SQL engine and the SQL engine can hurt performance.

Use the FORALL and BULK COLLECT INTO keywords together to bulk-bind the collection and improve performance:

```

DECLARE
    TYPE Emplist IS VARRAY(100) OF NUMBER;
    TYPE Numlist IS TABLE OF Emp_tab.Sal%TYPE;
    Empids EMPLIST := EMPLIST(7369, 7499, 7521, 7566, 7654, 7698);
    Bonlist NUMLIST;
BEGIN
    FORALL i IN Empids.FIRST..empIDs.LAST
        UPDATE Emp_tab SET Bonus = 0.1 * Sal
            WHERE Empno = Empids(i)
            RETURNING Sal BULK COLLECT INTO Bonlist;
```

```
END;
```

## Triggers

A trigger is a special kind of PL/SQL anonymous block. You can define triggers to fire before or after SQL statements, either on a statement level or for each row that is affected. You can also define `INSTEAD OF` triggers or system triggers (triggers on `DATABASE` and `SCHEMA`).

**See Also:** [Chapter 12, "Using Triggers"](#).

## Wrapping PL/SQL Code

You can deliver your stored procedures in object code format using the PL/SQL Wrapper. Wrapping your PL/SQL code hides your application internals. To run the PL/SQL Wrapper, enter the `WRAP` statement at your system prompt using the following syntax:

```
wrap INAME=input_file [ONAME=output_file]
```

**See Also:** For complete instructions on using the PL/SQL Wrapper, see the *PL/SQL User's Guide and Reference*.

## Remote Dependencies

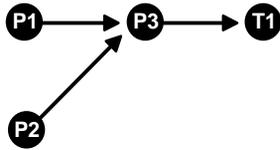
Dependencies among PL/SQL program units can be handled in two ways:

- [Timestamps](#)
- [Signatures](#)

### Timestamps

If timestamps are used to handle dependencies among PL/SQL program units, then whenever you alter a program unit or a relevant schema object, all of its dependent units are marked as invalid and must be recompiled before they can be run.

Each program unit carries a timestamp that is set by the server when the unit is created or recompiled. [Figure 9–1](#) demonstrates this graphically. Procedures `P1` and `P2` call stored procedure `P3`. Stored procedure `P3` references table `T1`. In this example, each of the procedures is dependent on table `T1`. `P3` depends upon `T1` directly, while `P1` and `P2` depend upon `T1` indirectly.

**Figure 9–1** *Dependency Relationships*

If P3 is altered, then P1 and P2 are marked as invalid immediately, if they are on the same server as P3. The compiled states of P1 and P2 contain records of the timestamp of P3. Therefore, if the procedure P3 is altered and recompiled, then the timestamp on P3 no longer matches the value that was recorded for P3 during the compilation of P1 and P2.

If P1 and P2 are on a client system, or on another Oracle Server in a distributed environment, then the timestamp information is used to mark them as invalid at runtime.

### **Disadvantages of the Timestamp Model**

The disadvantage of this dependency model is that it is unnecessarily restrictive. Recompile of dependent objects across the network are often performed when not strictly necessary, leading to performance degradation.

Furthermore, on the client side, the timestamp model can lead to situations that block an application from running at all, if the client-side application is built using PL/SQL version 2. Earlier releases of tools, such as Oracle Forms, that used PL/SQL version 1 on the client side did not use this dependency model, because PL/SQL version 1 had no support for stored procedures.

For releases of Oracle Forms that are integrated with PL/SQL version 2 on the client side, the timestamp model can present problems. For example, during the installation of the application, the application is rendered invalid unless the client-side PL/SQL procedures that it uses are recompiled at the client site. Also, if a client-side procedure depends on a server procedure, and if the server procedure is changed or automatically recompiled, then the client-side PL/SQL procedure must then be recompiled. Yet in many application environments (such as Forms runtime applications), there is no PL/SQL compiler available on the client. This blocks the application from running at all. The client application developer must then redistribute new versions of the application to all customers.

## Signatures

To alleviate some of the problems with the timestamp-only dependency model, Oracle provides the additional capability of remote dependencies using signatures. The signature capability affects only remote dependencies. Local (same server) dependencies are not affected, as recompilation is always possible in this environment.

A signature is associated with each compiled stored program unit. It identifies the unit using the following criteria:

- The name of the unit (the package, procedure, or function name).
- The types of each of the parameters of the subprogram.
- The modes of the parameters (IN, OUT, IN OUT).
- The number of parameters.
- The type of the return value for a function.

The user has control over whether signatures or timestamps govern remote dependencies.

**See Also:** ["Controlling Remote Dependencies"](#) on page 9-35.

When the signature dependency model is used, a dependency on a remote program unit causes an invalidation of the dependent unit if the dependent unit contains a call to a subprogram in the parent unit, and if the signature of this subprogram has been changed in an incompatible manner.

For example, consider a procedure `Get_emp_name` stored on a server in Boston (`BOSTON_SERVER`). The procedure is defined as the following:

---

---

**Note:** You may need to set up data structures, similar to the following, for certain examples to work:

```
CONNECT system/manager
CREATE PUBLIC DATABASE LINK boston_server USING 'inst1_alias';
CONNECT scott/tiger
```

---

---

```
CREATE OR REPLACE PROCEDURE Get_emp_name (
    emp_number    IN    NUMBER,
    hire_date     OUT  VARCHAR2,
    emp_name      OUT  VARCHAR2) AS
BEGIN
```

```

SELECT ename, to_char(hiredate, 'DD-MON-YY')
      INTO emp_name, hire_date
      FROM emp
      WHERE empno = emp_number;
END;

```

When `Get_emp_name` is compiled on `BOSTON_SERVER`, its signature, as well as its timestamp, is recorded.

Now, assume that on another server in California, some PL/SQL code calls `Get_emp_name` identifying it using a Dblink called `BOSTON_SERVER`, as follows:

```

CREATE OR REPLACE PROCEDURE print_ename (emp_number IN NUMBER) AS
  hire_date  VARCHAR2(12);
  ename      VARCHAR2(10);
BEGIN
  get_emp_name@BOSTON_SERVER(emp_number, hire_date, ename);
  dbms_output.put_line(ename);
  dbms_output.put_line(hire_date);
END;

```

When this California server code is compiled, the following actions take place:

- A connection is made to the Boston server.
- The signature of `Get_emp_name` is transferred to the California server.
- The signature is recorded in the compiled state of `Print_ename`.

At runtime, during the remote procedure call from the California server to the Boston server, the recorded signature of `Get_emp_name` that was saved in the compiled state of `Print_ename` gets sent to the Boston server, regardless of whether or not there were any changes.

If the timestamp dependency mode is in effect, then a mismatch in timestamps causes an error status to be returned to the calling procedure.

However, if the signature mode is in effect, then any mismatch in timestamps is ignored, and the recorded signature of `Get_emp_name` in the compiled state of `Print_ename` on the California server is compared with the current signature of `Get_emp_name` on the Boston server. If they match, then the call succeeds. If they do not match, then an error status is returned to the `Print_name` procedure.

Note that the `Get_emp_name` procedure on the Boston server could have been changed. Or, its timestamp could be different from that recorded in the `Print_name` procedure on the California server, possibly due to the installation of a new release of the server. As long as the signature remote dependency mode is in effect

on the California server, a timestamp mismatch does not cause an error when `Get_emp_name` is called.

---

---

**Note:** `DETERMINISTIC`, `PARALLEL_ENABLE`, and purity information do not show in the signature mode. Optimizations based on these settings are not automatically reconsidered if a function on a remote system is redefined with different settings. This may lead to incorrect query results when calls to the remote function occur, even indirectly, in a SQL statement, or if the remote function is used, even indirectly, in a function-based index.

---

---

## When Does a Signature Change?

**Datatypes** A signature changes when you switch from one class of datatype to another. Within each datatype class, there can be several types. Changing a parameter datatype from one type to another within a class does not cause the signature to change.

[Table 9–3](#) lists the classes of types.

**Table 9–3** *Datatypes*

<b>Varchar Types</b>	<b>Number Types</b>
VARCHAR2	NUMBER
VARCHAR	INTEGER
STRING	INT
LONG	SMALLINT
ROWID	DECIMAL
<b>Character Types</b>	DEC
CHARACTER	REAL
CHAR	FLOAT
<b>Raw Types</b>	NUMERIC
RAW	DOUBLE PRECISION
LONG RAW	NUMERIC
<b>Integer Types</b>	<b>Date Type</b>

**Table 9–3 Datatypes**

BINARY_INTEGER	DATE
PLS_INTEGER	
BOOLEAN	
NATURAL	
POSITIVE	
POSITIVEN	
NATURALN	

---

**Modes** Changing to or from an explicit specification of the default parameter mode `IN` does not change the signature of a subprogram. For example, you change

```
PROCEDURE P1 (Param1 NUMBER);
```

to

```
PROCEDURE P1 (Param1 IN NUMBER);
```

This does not change the signature. Any other change of parameter mode *does* change the signature.

**Default Parameter Values** Changing the specification of a default parameter value does not change the signature. For example, procedure `P1` has the same signature in the following two examples:

```
PROCEDURE P1 (Param1 IN NUMBER := 100);
PROCEDURE P1 (Param1 IN NUMBER := 200);
```

An application developer who requires that callers get the new default value must recompile the called procedure, but no signature-based invalidation occurs when a default parameter value assignment is changed.

## Examples of Signatures

Using the `Get_emp_names` procedure defined in "[Parameters for Procedures and Functions](#)" on page 9-5, if the procedure body is changed to the following:

```
DECLARE
    Emp_number NUMBER;
    Hire_date DATE;
BEGIN
```

```
-- date format model changes

SELECT Ename, To_char(Hiredate, 'DD/MON/YYYY')
       INTO Emp_name, Hire_date
FROM Emp_tab
WHERE Empno = Emp_number;

END;
```

Then, the specification of the procedure has not changed, and, therefore, its signature has not changed.

But, if the procedure specification is changed to the following:

```
CREATE OR REPLACE PROCEDURE Get_emp_name (
    Emp_number IN NUMBER,
    Hire_date  OUT DATE,
    Emp_name   OUT VARCHAR2) AS
```

And, if the body is changed accordingly, then the signature changes, because the parameter `Hire_date` has a different datatype.

However, if the name of that parameter changes to `When_hired`, and the datatype remains `VARCHAR2`, and the mode remains `OUT`, then the signature does *not* change. Changing the *name* of a formal parameter does not change the signature of the unit.

Consider the following example:

```
CREATE OR REPLACE PACKAGE Emp_package AS
    TYPE Emp_data_type IS RECORD (
        Emp_number NUMBER,
        Hire_date  VARCHAR2(12),
        Emp_name   VARCHAR2(10));
    PROCEDURE Get_emp_data
        (Emp_data IN OUT Emp_data_type);
END;

CREATE OR REPLACE PACKAGE BODY Emp_package AS
    PROCEDURE Get_emp_data
        (Emp_data IN OUT Emp_data_type) IS
    BEGIN
        SELECT Empno, Ename, TO_CHAR(Hiredate, 'DD/MON/YY')
           INTO Emp_data
        FROM Emp_tab
        WHERE Empno = Emp_data.Emp_number;
    END;
END;
```

If the package specification is changed so that the record's field names are changed, but the types remain the same, then this does not affect the signature. For example, the following package specification has the same signature as the previous package specification example:

```
CREATE OR REPLACE PACKAGE Emp_package AS
  TYPE Emp_data_type IS RECORD (
    Emp_num    NUMBER,          -- was Emp_number
    Hire_dat   VARCHAR2(12),   -- was Hire_date
    Empname    VARCHAR2(10));  -- was Emp_name
  PROCEDURE Get_emp_data
    (Emp_data IN OUT Emp_data_type);
END;
```

Changing the name of the type of a parameter does not cause a change in the signature if the type remains the same as before. For example, the following package specification for `Emp_package` is the same as the first one:

```
CREATE OR REPLACE PACKAGE Emp_package AS
  TYPE Emp_data_record_type IS RECORD (
    Emp_number NUMBER,
    Hire_date  VARCHAR2(12),
    Emp_name   VARCHAR2(10));
  PROCEDURE Get_emp_data
    (Emp_data IN OUT Emp_data_record_type);
END;
```

## Controlling Remote Dependencies

The dynamic initialization parameter `REMOTE_DEPENDENCIES_MODE` controls whether the timestamp or the signature dependency model is in effect.

- If the initialization parameter file contains the following specification:

```
REMOTE_DEPENDENCIES_MODE = TIMESTAMP
```

Then only timestamps are used to resolve dependencies (if this is not explicitly overridden dynamically).

- If the initialization parameter file contains the following parameter specification:

```
REMOTE_DEPENDENCIES_MODE = SIGNATURE
```

Then signatures are used to resolve dependencies (if this not explicitly overridden dynamically).

- You can alter the mode dynamically by using the DDL statements. For example:

```
ALTER SESSION SET REMOTE_DEPENDENCIES_MODE =  
    {SIGNATURE | TIMESTAMP}
```

The above example alters the dependency model for the current session.

```
ALTER SYSTEM SET REMOTE_DEPENDENCIES_MODE =  
    {SIGNATURE | TIMESTAMP}
```

The above example alters the dependency model on a system-wide basis after startup.

If the `REMOTE_DEPENDENCIES_MODE` parameter is not specified, either in the `init.ora` parameter file or using the `ALTER SESSION` or `ALTER SYSTEM` DDL statements, then timestamp is the default value. Therefore, unless you explicitly use the `REMOTE_DEPENDENCIES_MODE` parameter, or the appropriate DDL statement, your server is operating using the timestamp dependency model.

When you use `REMOTE_DEPENDENCIES_MODE=SIGNATURE`, you should be aware of the following:

- If you change the default value of a parameter of a remote procedure, then the local procedure calling the remote procedure is not invalidated. If the call to the remote procedure does not supply the parameter, then the default value is used. In this case, because invalidation/recompilation does not automatically occur, the old default value is used. If you want to see the new default values, then you must recompile the calling procedure manually.
- If you add a new overloaded procedure in a package (a new procedure with the same name as an existing one), then local procedures that call the remote procedure are not invalidated. If it turns out that this overloading results in a rebinding of existing calls from the local procedure under the timestamp mode, then this rebinding does not happen under the signature mode, because the local procedure does not get invalidated. You must recompile the local procedure manually to achieve the new rebinding.
- If the types of parameters of an existing packaged procedure are changed so that the new types have the same shape as the old ones, then the local calling procedure is not invalidated or recompiled automatically. You must recompile the calling procedure manually to get the semantics of the new type.

### Dependency Resolution

When `REMOTE_DEPENDENCIES_MODE = TIMESTAMP` (the default value), dependencies among program units are handled by comparing timestamps at

runtime. If the timestamp of a called remote procedure does not match the timestamp of the called procedure, then the calling (dependent) unit is invalidated and must be recompiled. In this case, if there is no local PL/SQL compiler, then the calling application cannot proceed.

In the timestamp dependency mode, signatures are not compared. If there is a local PL/SQL compiler, then recompilation happens automatically when the calling procedure is run.

When `REMOTE_DEPENDENCIES_MODE = SIGNATURE`, the recorded timestamp in the calling unit is first compared to the current timestamp in the called remote unit. If they match, then the call proceeds. If the timestamps do not match, then the signature of the called remote subprogram, as recorded in the calling subprogram, is compared with the current signature of the called subprogram. If they do not match (using the criteria described in the section "[When Does a Signature Change?](#)" on page 9-32), then an error is returned to the calling session.

### Suggestions for Managing Dependencies

Oracle recommends that you follow these guidelines for setting the `REMOTE_DEPENDENCIES_MODE` parameter:

- Server-side PL/SQL users can set the parameter to `TIMESTAMP` (or let it default to that) to get the timestamp dependency mode.
- Server-side PL/SQL users can choose to use the signature dependency mode if they have a distributed system and they want to avoid possible unnecessary recompilations.
- Client-side PL/SQL users should set the parameter to `SIGNATURE`. This allows:
  - Installation of new applications at client sites, without the need to recompile procedures.
  - Ability to upgrade the server, without encountering timestamp mismatches.
- When using signature mode on the server side, add new procedures to the end of the procedure (or function) declarations in a package specification. Adding a new procedure in the middle of the list of declarations can cause unnecessary invalidation and recompilation of dependent procedures.

## Cursor Variables

A cursor is a static object; a cursor variable is a pointer to a cursor. Because cursor variables are pointers, they can be passed and returned as parameters to procedures and functions. A cursor variable can also refer to different cursors in its lifetime.

Some additional advantages of cursor variables include:

- **Encapsulation** Queries are centralized in the stored procedure that opens the cursor variable.
- **Ease of maintenance** If you need to change the cursor, then you only need to make the change in one place: the stored procedure. There is no need to change each application.
- **Convenient security** The user of the application is the username used when the application connects to the server. The user must have `EXECUTE` permission on the stored procedure that opens the cursor. But, the user does not need to have `READ` permission on the tables used in the query. This capability can be used to limit access to the columns in the table, as well as access to other stored procedures.

**See Also:** The *PL/SQL User's Guide and Reference* has a complete discussion of cursor variables.

## Declaring and Opening Cursor Variables

Memory is usually allocated for a cursor variable in the client application using the appropriate `ALLOCATE` statement. In Pro\*C, use the `EXEC SQL ALLOCATE <cursor_name>` statement. In OCI, use the Cursor Data Area.

You can also use cursor variables in applications that run entirely in a single server session. You can declare cursor variables in PL/SQL subprograms, open them, and use them as parameters for other PL/SQL subprograms.

## Examples of Cursor Variables

This section includes several examples of cursor variable usage in PL/SQL. For additional cursor variable examples that use the programmatic interfaces, see the following manuals:

- *Pro\*C/C++ Precompiler Programmer's Guide*
- *Pro\*COBOL Precompiler Programmer's Guide*
- *Oracle Call Interface Programmer's Guide*
- *SQL\*Module for Ada Programmer's Guide*

## Fetching Data

The following package defines a PL/SQL cursor variable type `Emp_val_cv_type`, and two procedures. The first procedure, `Open_emp_cv`, opens the cursor variable using a bind variable in the `WHERE` clause. The second procedure, `Fetch_emp_data`, fetches rows from the `Emp_tab` table using the cursor variable.

```
CREATE OR REPLACE PACKAGE Emp_data AS
  TYPE Emp_val_cv_type IS REF CURSOR RETURN Emp_tab%ROWTYPE;
  PROCEDURE Open_emp_cv (Emp_cv          IN OUT Emp_val_cv_type,
                        Dept_number      IN      INTEGER);
  PROCEDURE Fetch_emp_data (emp_cv       IN      Emp_val_cv_type,
                            emp_row      OUT      Emp_tab%ROWTYPE);
END Emp_data;

CREATE OR REPLACE PACKAGE BODY Emp_data AS
  PROCEDURE Open_emp_cv (Emp_cv          IN OUT Emp_val_cv_type,
                        Dept_number      IN      INTEGER) IS
  BEGIN
    OPEN emp_cv FOR SELECT * FROM Emp_tab WHERE deptno = dept_number;
  END open_emp_cv;
  PROCEDURE Fetch_emp_data (Emp_cv       IN      Emp_val_cv_type,
                            Emp_row      OUT      Emp_tab%ROWTYPE) IS
  BEGIN
    FETCH Emp_cv INTO Emp_row;
  END Fetch_emp_data;
END Emp_data;
```

The following example shows how to call the `Emp_data` package procedures from a PL/SQL block:

```
DECLARE
-- declare a cursor variable
  Emp_curs Emp_data.Emp_val_cv_type;
  Dept_number Dept_tab.Deptno%TYPE;
  Emp_row Emp_tab%ROWTYPE;

BEGIN
  Dept_number := 20;
-- open the cursor using a variable
  Emp_data.Open_emp_cv(Emp_curs, Dept_number);
-- fetch the data and display it
  LOOP
    Emp_data.Fetch_emp_data(Emp_curs, Emp_row);
    EXIT WHEN Emp_curs%NOTFOUND;
    DBMS_OUTPUT.PUT(Emp_row.Ename || ' ');
    DBMS_OUTPUT.PUT_LINE(Emp_row.Sal);
  END LOOP;
END;
```

## Implementing Variant Records

The power of cursor variables comes from their ability to point to different cursors. In the following package example, a discriminant is used to open a cursor variable to point to one of two different cursors:

```
CREATE OR REPLACE PACKAGE Emp_dept_data AS
  TYPE Cv_type IS REF CURSOR;
  PROCEDURE Open_cv (Cv          IN OUT cv_type,
                    Discrim      IN      POSITIVE);
END Emp_dept_data;

CREATE OR REPLACE PACKAGE BODY Emp_dept_data AS
  PROCEDURE Open_cv (Cv          IN OUT cv_type,
                    Discrim      IN      POSITIVE) IS
  BEGIN
    IF Discrim = 1 THEN
      OPEN Cv FOR SELECT * FROM Emp_tab WHERE Sal > 2000;
    ELSIF Discrim = 2 THEN
      OPEN Cv FOR SELECT * FROM Dept_tab;
    END IF;
  END Open_cv;
END Emp_dept_data;
```

You can call the `Open_cv` procedure to open the cursor variable and point it to either a query on the `Emp_tab` table or the `Dept_tab` table. The following PL/SQL

block shows how to fetch using the cursor variable, and then use the ROWTYPE\_MISMATCH predefined exception to handle either fetch:

```

DECLARE
    Emp_rec  Emp_tab%ROWTYPE;
    Dept_rec Dept_tab%ROWTYPE;
    Cv       Emp_dept_data.CV_TYPE;

BEGIN
    Emp_dept_data.open_cv(Cv, 1); -- Open Cv For Emp_tab Fetch
    Fetch cv INTO Dept_rec;      -- but fetch into Dept_tab record
                                -- which raises ROWTYPE_MISMATCH

    DBMS_OUTPUT.PUT(Dept_rec.Deptno);
    DBMS_OUTPUT.PUT_LINE(' ' || Dept_rec.Loc);

EXCEPTION
    WHEN ROWTYPE_MISMATCH THEN
        BEGIN
            DBMS_OUTPUT.PUT_LINE
                ('Row type mismatch, fetching Emp_tab data...');
            FETCH Cv INTO Emp_rec;
            DBMS_OUTPUT.PUT(Emp_rec.Deptno);
            DBMS_OUTPUT.PUT_LINE(' ' || Emp_rec.Ename);
        END;

```

## Compile-Time Errors

When you use SQL\*Plus to submit PL/SQL code, and when the code contains errors, you receive notification that compilation errors have occurred, but there is no immediate indication of what the errors are. For example, if you submit a stand-alone (or stored) procedure PROC1 in the file proc1.sql as follows:

```
SQL> @proc1
```

And, if there are one or more errors in the code, then you receive a notice such as the following:

```
MGR-00072: Warning: Procedure proc1 created with compilation errors
```

In this case, use the SHOW ERRORS statement in SQL\*Plus to get a list of the errors that were found. SHOW ERRORS with no argument lists the errors from the most recent compilation. You can qualify SHOW ERRORS using the name of a procedure, function, package, or package body:

```
SQL> SHOW ERRORS PROC1
SQL> SHOW ERRORS PROCEDURE PROC1
```

**See Also:** See the *SQL\*Plus User's Guide and Reference* for complete information about the SHOW ERRORS statement.

---

---

**Note:** Before issuing the SHOW ERRORS statement, use the SET CHARWIDTH statement to get long lines on output. The value 132 is usually a good choice. For example:

```
SET CHARWIDTH 132
```

---

---

Assume that you want to create a simple procedure that deletes records from the employee table using SQL\*Plus:

```
CREATE OR REPLACE PROCEDURE Fire_emp(Emp_id NUMBER) AS
BEGIN
    DELETE FROM Emp_tab WHERE Empno = Emp_id;
END
/
```

Notice that the CREATE PROCEDURE statement has two errors: the DELETE statement has an error (the 'E' is absent from WHERE), and the semicolon is missing after END.

After the CREATE PROCEDURE statement is entered and an error is returned, a SHOW ERRORS statement returns the following lines:

```
SHOW ERRORS;

ERRORS FOR PROCEDURE Fire_emp:
LINE/COL      ERROR
-----
3/27          PL/SQL-00103: Encountered the symbol "EMPNO" wh. . .
5/0           PL/SQL-00103: Encountered the symbol "END" when . . .
2 rows selected.
```

Notice that each line and column number where errors were found is listed by the SHOW ERRORS statement.

Alternatively, you can query the following data dictionary views to list errors when using any tool or application:

- USER\_ERRORS

- ALL\_ERRORS
- DBA\_ERRORS

The error text associated with the compilation of a procedure is updated when the procedure is replaced, and it is deleted when the procedure is dropped.

Original source code can be retrieved from the data dictionary using the following views: ALL\_SOURCE, USER\_SOURCE, and DBA\_SOURCE.

**See Also:** *Oracle8i Reference* for more information about these data dictionary views.

## Run-Time Error Handling

Oracle allows user-defined errors in PL/SQL code to be handled so that user-specified error numbers and messages are returned to the client application. After received, the client application can handle the error based on the user-specified error number and message returned by Oracle.

User-specified error messages are returned using the RAISE\_APPLICATION\_ERROR procedure. For example:

```
RAISE_APPLICATION_ERROR(Error_number, 'text', Keep_error_stack)
```

This procedure stops procedure execution, rolls back any effects of the procedure, and returns a user-specified error number and message (unless the error is trapped by an exception handler). ERROR\_NUMBER must be in the range of -20000 to -20999.

Error number -20000 should be used as a generic number for messages where it is important to relay information to the user, but having a unique error number is not required. Text must be a character expression, 2 Kbytes or less (longer messages are ignored). Keep\_error\_stack can be TRUE if you want to add the error to any already on the stack, or FALSE if you want to replace the existing errors. By default, this option is FALSE.

---

---

**Note:** Some of the Oracle-supplied packages, such as DBMS\_OUTPUT, DBMS\_DESCRIBE, and DBMS\_ALERT, use application error numbers in the range -20000 to -20005. See the descriptions of these packages for more information.

---

---

The RAISE\_APPLICATION\_ERROR procedure is often used in exception handlers or in the logic of PL/SQL code. For example, the following exception handler

selects the string for the associated user-defined error message and calls the `RAISE_APPLICATION_ERROR` procedure:

```
...  
WHEN NO_DATA_FOUND THEN  
    SELECT Error_string INTO Message  
    FROM Error_table,  
    V$NLS_PARAMETERS V  
    WHERE Error_number = -20101 AND Lang = v.value AND  
          v.parameter = "NLS_LANGUAGE";  
    Raise_application_error(-20101, Message);  
...
```

**See Also:** For information on exception handling when calling remote procedures, see "[Handling Errors in Remote Procedures](#)" on page 9-46.

The following section includes an example of passing a user-specified error number from a trigger to a procedure.

## Declaring Exceptions and Exception Handling Routines

User-defined exceptions are explicitly defined and signaled within the PL/SQL block to control processing of errors specific to the application. When an exception is *raised* (signaled), the usual execution of the PL/SQL block stops, and a routine called an exception handler is called. Specific exception handlers can be written to handle any internal or user-defined exception.

Application code can check for a condition that requires special attention using an `IF` statement. If there is an error condition, then two options are available:

- Enter a `RAISE` statement that names the appropriate exception. A `RAISE` statement stops the execution of the procedure, and control passes to an exception handler (if any).
- Call the `RAISE_APPLICATION_ERROR` procedure to return a user-specified error number and message.

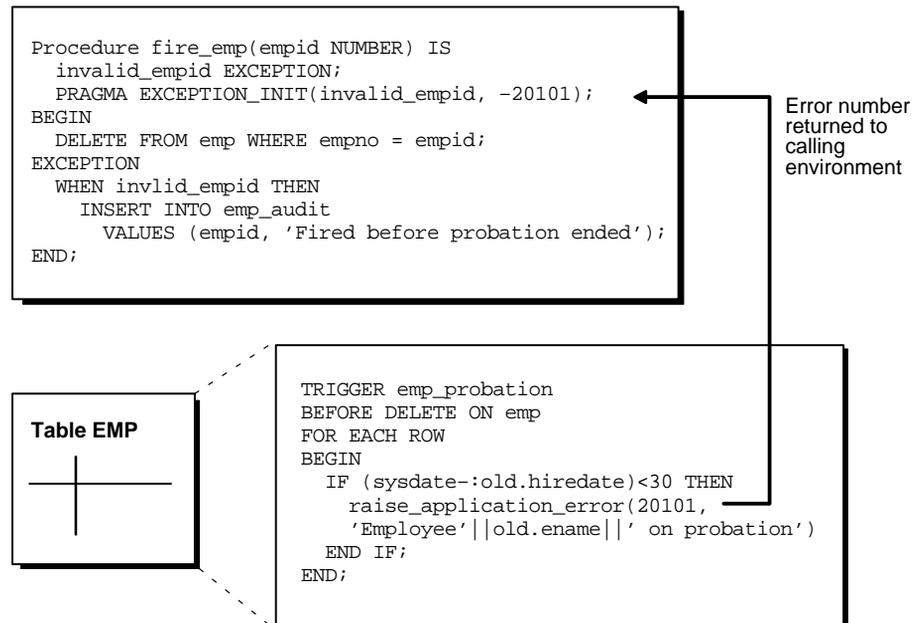
You can also define an exception handler to handle user-specified error messages. For example, [Figure 9-2](#) on page 9-45 illustrates the following:

- An exception and associated exception handler in a procedure
- A conditional statement that checks for an error (such as transferring funds not available) and enters a user-specified error number and message within a trigger

- How user-specified error numbers are returned to the calling environment (in this case, a procedure), and how that application can define an exception that corresponds to the user-specified error number

*Declare* a user-defined exception in a procedure or package body (private exceptions), or in the specification of a package (public exceptions). *Define* an exception handler in the body of a procedure (stand-alone or package).

**Figure 9–2 Exceptions and User-Defined Errors**



## Unhandled Exceptions

In database PL/SQL program units, an unhandled user-error condition or internal error condition that is not trapped by an appropriate exception handler causes the implicit rollback of the program unit. If the program unit includes a `COMMIT` statement before the point at which the unhandled exception is observed, then the implicit rollback of the program unit can only be completed back to the previous `COMMIT`.

Additionally, unhandled exceptions in database-stored PL/SQL program units propagate back to client-side applications that call the containing program unit. In such an application, only the application program unit call is rolled back (not the entire application program unit), because it is submitted to the database as a SQL statement.

If unhandled exceptions in database PL/SQL program units are propagated back to database applications, then the database PL/SQL code should be modified to handle the exceptions. Your application can also trap for unhandled exceptions when calling database program units and handle such errors appropriately.

## Handling Errors in Distributed Queries

You can use a trigger or a stored procedure to create a distributed query. This distributed query is decomposed by the local Oracle into a corresponding number of remote queries, which are sent to the remote nodes for execution. The remote nodes run the queries and send the results back to the local node. The local node then performs any necessary post-processing and returns the results to the user or application.

If a portion of a distributed statement fails, possibly due to an integrity constraint violation, then Oracle returns error number `ORA-02055`. Subsequent statements, or procedure calls, return error number `ORA-02067` until a rollback or a rollback to savepoint is entered.

You should design your application to check for any returned error messages that indicates that a portion of the distributed update has failed. If you detect a failure, then you should rollback the entire transaction (or rollback to a savepoint) before allowing the application to proceed.

## Handling Errors in Remote Procedures

When a procedure is run locally or at a remote location, four types of exceptions can occur:

- PL/SQL user-defined exceptions, which must be declared using the keyword `EXCEPTION`.
- PL/SQL predefined exceptions, such as `NO_DATA_FOUND`.
- SQL errors, such as `ORA-00900` and `ORA-02015`.
- Application exceptions, which are generated using the `RAISE_APPLICATION_ERROR()` procedure.

When using local procedures, all of these messages can be trapped by writing an exception handler, such as shown in the following example:

```
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    /* ...handle the exception */
```

Notice that the `WHEN` clause requires an exception name. If the exception that is raised does not have a name, such as those generated with `RAISE_APPLICATION_ERROR`, then one can be assigned using `PRAGMA_EXCEPTION_INIT`, as shown in the following example:

```
DECLARE
  ...
  Null_salary EXCEPTION;
  PRAGMA EXCEPTION_INIT(Null_salary, -20101);
BEGIN
  ...
  RAISE_APPLICATION_ERROR(-20101, 'salary is missing');
  ...
EXCEPTION
  WHEN Null_salary THEN
    ...
```

When calling a remote procedure, exceptions are also handled by creating a local exception handler. The remote procedure must return an error number to the local calling procedure, which then handles the exception, as shown in the previous example. Because PL/SQL user-defined exceptions always return `ORA-06510` to the local procedure, these exceptions cannot be handled. All other remote exceptions can be handled in the same manner as local exceptions.

## Debugging Stored Procedures

A free Java-based debugger is available from Oracle for debugging PL/SQL applications. This is a lightweight debugger which can be run as an applet from the Microsoft or Netscape browser, or as a stand-alone application to debug PL/SQL stored procedures. The debugger works with Oracle 7.3.4 and higher.

If you are using PL/SQL for writing Web-based applications in conjunction with the Applications Server, and if you would like to debug and look at the generated HTML, you can use this debugger as well (check restrictions in the `README` file).

You can download the debugger from:

<http://www.oracle.com/st/products/features/plsql/plsqlfreeutility.html>

The `DBMS_DEBUG` API, provided with Oracle8i, implements server-side debuggers, and it provides a way to debug server-side PL/SQL program units. Several of the debuggers currently available, such as Oracle Procedure Builder and various third-party vendor solutions, use this API.

Oracle Procedure Builder is an advanced client-server debugger that transparently debugs your database applications. It lets you run PL/SQL procedures and triggers in a controlled debugging environment, and you can set breakpoints, list the values of variables, and perform other debugging tasks. Oracle Procedure Builder is part of the Oracle Developer tool set.

**See Also:** *Oracle Procedure Builder Developer's Guide*

You can also debug stored procedures and triggers using the `DBMS_OUTPUT` supplied package. Put `PUT` and `PUT_LINE` statements in your code to output the value of variables and expressions to your terminal.

**See Also:** See *Oracle8i Supplied PL/SQL Packages Reference* for more information about the `DBMS_DEBUG` and the `DBMS_OUTPUT` packages.

## Calling Stored Procedures

---



---

**Note:** You may need to set up data structures, similar to the following, for certain examples to work:

```
CREATE TABLE Emp_tab (
  Empno   NUMBER(4) NOT NULL,
  Ename   VARCHAR2(10),
  Job     VARCHAR2(9),
  Mgr     NUMBER(4),
  Hiredate DATE,
  Sal     NUMBER(7,2),
  Comm    NUMBER(7,2),
  Deptno  NUMBER(2));

CREATE OR REPLACE PROCEDURE fire_emp1(Emp_id NUMBER) AS
BEGIN
  DELETE FROM Emp_tab WHERE Empno = Emp_id;
END;
VARIABLE Empnum NUMBER;
```

---



---

Procedures can be called from many different environments. For example:

- A procedure can be called within the body of another procedure or a trigger.
- A procedure can be interactively called by a user using an Oracle tool.
- A procedure can be explicitly called within an application, such as a SQL\*Forms or a precompiler application.
- A stored function can be called from a SQL statement in a manner similar to calling a built-in SQL function, such as LENGTH or ROUND.

This section includes some common examples of calling procedures from within these environments.

**See Also:** ["Calling Stored Functions from SQL Expressions"](#) on page 9-56.

### A Procedure or Trigger Calling Another Procedure

A procedure or trigger can call another stored procedure. For example, included in the body of one procedure might be the following line:

```
. . .
Sal_raise(Emp_id, 200);
```

. . .

This line calls the `Sal_raise` procedure. `Emp_id` is a variable within the context of the procedure. Recursive procedure calls are allowed within PL/SQL: A procedure can call itself.

### Interactively Calling Procedures From Oracle Tools

A procedure can be called interactively from an Oracle tool, such as SQL\*Plus. For example, to call a procedure named `SAL_RAISE`, owned by you, you can use an anonymous PL/SQL block, as follows:

```
BEGIN
    Sal_raise(7369, 200);
END;
```

---

---

**Note:** Interactive tools, such as SQL\*Plus, require you to follow these lines with a slash (/) to run the PL/SQL block.

---

---

An easier way to run a block is to use the SQL\*Plus statement `EXECUTE`, which wraps `BEGIN` and `END` statements around the code you enter. For example:

```
EXECUTE Sal_raise(7369, 200);
```

Some interactive tools allow session variables to be created. For example, when using SQL\*Plus, the following statement creates a session variable:

```
VARIABLE Assigned_empno NUMBER
```

After defined, any session variable can be used for the duration of the session. For example, you might run a function and capture the return value using a session variable:

```
EXECUTE :Assigned_empno := Hire_emp('JSMITH', 'President',
    1032, SYSDATE, 5000, NULL, 10);
PRINT Assigned_empno;
ASSIGNED_EMPNO
-----
          2893
```

**See Also:** See the *SQL\*Plus User's Guide and Reference* for SQL\*Plus information. See your tools documentation for information about performing similar operations using your development tool.

## Calling Procedures within 3GL Applications

A 3GL database application, such as a precompiler or an OCI application, can include a call to a procedure within the code of the application.

To run a procedure within a PL/SQL block in an application, simply call the procedure. The following line within a PL/SQL block calls the `Fire_emp` procedure:

```
Fire_emp1(:Empnum);
```

In this case, `:Empno` is a host (bind) variable within the context of the application.

To run a procedure within the code of a precompiler application, you must use the `EXEC` call interface. For example, the following statement calls the `Fire_emp` procedure in the code of a precompiler application:

```
EXEC SQL EXECUTE
  BEGIN
    Fire_emp1(:Empnum);
  END;
END-EXEC;
```

**See Also:** For more information about calling PL/SQL procedures from within 3GL applications, see the following manuals:

*Oracle Call Interface Programmer's Guide*

*Pro\*C/C++ Precompiler Programmer's Guide*,

*SQL\*Module for Ada Programmer's Guide*

## Name Resolution When Calling Procedures

References to procedures and packages are resolved according to the algorithm described in the ["Name Resolution in SQL Statements"](#) section of [Chapter 2](#), ["Managing Schema Objects"](#).

### Privileges Required to Execute a Procedure

If you are the owner of a stand-alone procedure or package, then you can run the stand-alone procedure or packaged procedure, or any public procedure or packaged procedure at any time, as described in the previous sections. If you want to run a stand-alone or packaged procedure owned by another user, then the following conditions apply:

- You must have the `EXECUTE` privilege for the stand-alone procedure or package containing the procedure, or you must have the `EXECUTE ANY PROCEDURE` system privilege. If you are executing a remote procedure, then you must be granted the `EXECUTE` privilege or `EXECUTE ANY PROCEDURE` system privilege directly, not through a role.
- You must include the owner's name in the call. For example:

---



---

**Note:** You may need to set up the following data structures for certain examples to work:

```
CONNECT sys/change_on_install AS Sysdba;
CREATE USER Jward IDENTIFIED BY Jward;
GRANT CREATE ANY PACKAGE TO Jward;
GRANT CREATE SESSION TO Jward;
GRANT EXECUTE ANY PROCEDURE TO Jward;
CONNECT Scott/Tiger
```

---



---

```
EXECUTE Jward.Fire_emp (1043);
```

```
EXECUTE Jward.Hire_fire.Fire_emp (1043);
```

---



---

**Note:** A stored subprogram or package runs in the privilege domain of the owner of the procedure. The owner must be explicitly granted the necessary object privileges to all objects referenced within the body of the code.

---



---

### Specifying Values for Procedure Arguments

When you call a procedure, specify a value or parameter for each of the procedure's arguments. Identify the argument values using either of the following methods, or a combination of both:

- List the values in the order the arguments appear in the procedure declaration.
- Specify the argument names and corresponding values, in any order.

For example, these statements each call the procedure `Sal_raise` to increase the salary of employee number 7369 by 500:

```
Sal_raise(7369, 500);

Sal_raise(Sal_incr=>500, Emp_id=>7369);

Sal_raise(7369, Sal_incr=>500);
```

The first statement identifies the argument values by listing them in the order in which they appear in the procedure specification.

The second statement identifies the argument values by name and in an order different from that of the procedure specification. If you use argument names, then you can list the arguments in any order.

The third statement identifies the argument values using a combination of these methods. If you use a combination of order and argument names, then values identified in order must precede values identified by name.

If you used the `DEFAULT` option to define default values for `IN` parameters to a subprogram (see the *PL/SQL User's Guide and Reference*), then you can pass different numbers of actual parameters to the first subprogram, accepting or overriding the default values as you please. If an actual value is not passed, then the corresponding default value is used. If you want to assign a value to an argument that occurs after an omitted argument (for which the corresponding default is used), then you must explicitly designate the name of the argument, as well as its value.

## Calling Remote Procedures

Call remote procedures using an appropriate database link and the procedure's name. The following `SQL*Plus` statement runs the procedure `Fire_emp` located in the database and pointed to by the local database link named `BOSTON_SERVER`:

```
EXECUTE fire_emp1@boston_server(1043);
```

**See Also:** For information on exception handling when calling remote procedures, see "[Handling Errors in Remote Procedures](#)" on page 9-46.

### Remote Procedure Calls and Parameter Values

You must explicitly pass values to all remote procedure parameters, even if there are defaults. You cannot access remote package variables and constants.

## Referencing Remote Objects

Remote objects can be referenced within the body of a locally defined procedure. The following procedure deletes a row from the remote employee table:

```
CREATE OR REPLACE PROCEDURE fire_emp(emp_id NUMBER) IS
BEGIN
    DELETE FROM emp@boston_server WHERE empno = emp_id;
END;
```

The list below explains how to properly call remote procedures, depending on the calling environment.

- Remote procedures (stand-alone and packaged) can be called from within a procedure, an OCI application, or a precompiler application by specifying the remote procedure name, a database link, and the arguments for the remote procedure.

```
CREATE OR REPLACE PROCEDURE local_procedure(arg IN NUMBER) AS
BEGIN
    fire_empl@boston_server(arg);
END;
```

- In the previous example, you could create a synonym for FIRE\_EMP1@BOSTON\_SERVER. This would enable you to call the remote procedure from an Oracle tool application, such as a SQL\*Forms application, as well from within a procedure, OCI application, or precompiler application.

```
CREATE SYNONYM synonym1 for fire_empl@boston_server;
CREATE OR REPLACE PROCEDURE local_procedure(arg IN NUMBER) AS
BEGIN
    synonym1(arg);
END;
```

- If you do not want to use a synonym, then you could write a local cover procedure to call the remote procedure.

```
DECLARE
    arg NUMBER;
BEGIN
    local_procedure(arg);
END;
```

Here, `local_procedure` is defined as in the first item of this list.

---

**See Also:** ["Synonyms for Procedures and Packages"](#) on page 9-56

---

---

**Caution:** Unlike stored procedures, which use compile-time binding, runtime binding is used when referencing remote procedures. The user account to which you connect depends on the database link.

---

---

All calls to remotely stored procedures are assumed to perform updates; therefore, this type of referencing always requires two-phase commit of that transaction (even if the remote procedure is read-only). Furthermore, if a transaction that includes a remote procedure call is rolled back, then the work done by the remote procedure is also rolled back.

A procedure called remotely can usually execute a `COMMIT`, `ROLLBACK`, or `SAVEPOINT` statement, the same as a local procedure. However, there are some differences in behavior:

- If the transaction was originated by a non-Oracle database, as may be the case in XA applications, these operations are not allowed in the remote procedure.
- After doing one of these operations, the remote procedure cannot start any distributed transactions of its own.
- If the remote procedure does not commit or roll back its work, the commit is done implicitly when the database link is closed. In the meantime, further calls to the remote procedure are not allowed because it is still considered to be performing a transaction.

A *distributed update* modifies data on two or more nodes. A distributed update is possible using a procedure that includes two or more remote updates that access data on different nodes. Statements in the construct are sent to the remote nodes, and the execution of the construct succeeds or fails as a unit. If part of a distributed update fails and part succeeds, then a rollback (of the entire transaction or to a savepoint) is required to proceed. Consider this when creating procedures that perform distributed updates.

Pay special attention when using a local procedure that calls a remote procedure. If a timestamp mismatch is found during execution of the local procedure, then the remote procedure is not run, and the local procedure is invalidated.

## Synonyms for Procedures and Packages

Synonyms can be created for stand-alone procedures and packages to do the following:

- Hide the identity of the name and owner of a procedure or package.
- Provide location transparency for remotely stored procedures (stand-alone or within a package).

When a privileged user needs to call a procedure, an associated synonym can be used. Because the procedures defined within a package are not individual objects (the package is the object), synonyms cannot be created for individual procedures within a package.

## Calling Stored Functions from SQL Expressions

You can include user-written PL/SQL functions in SQL expressions. (You must be using PL/SQL release 2.1 or higher.) By using PL/SQL functions in SQL statements, you can do the following:

- Increase user productivity by extending SQL. Expressiveness of the SQL statement increases where activities are too complex, too awkward, or unavailable with SQL.
- Increase query efficiency. Functions used in the `WHERE` clause of a query can filter data using criteria that would otherwise need to be evaluated by the application.
- Manipulate character strings to represent special datatypes (for example, latitude, longitude, or temperature).
- Provide parallel query execution: If the query is parallelized, then SQL statements in your PL/SQL function may also be run in parallel (using the parallel query option).

## Using PL/SQL Functions

PL/SQL functions must be created as top-level functions or declared within a package specification before they can be named within a SQL statement. Stored PL/SQL functions are used in the same manner as built-in Oracle functions (such as `SUBSTR` or `ABS`).

PL/SQL functions can be placed wherever an Oracle function can be placed within a SQL statement, or, wherever expressions can occur in SQL. For example, they can be called from the following:

- The select list of the `SELECT` statement.
- The condition of the `WHERE` and `HAVING` clause.
- The `CONNECT BY`, `START WITH`, `ORDER BY`, and `GROUP BY` clauses.
- The `VALUES` clause of the `INSERT` statement.
- The `SET` clause of the `UPDATE` statement.

You cannot call stored PL/SQL functions from a `CHECK` constraint clause of a `CREATE` or `ALTER TABLE` statement or use them to specify a default value for a column. These situations require an unchanging definition.

---



---

**Note:** Unlike functions, which are called as part of an expression, procedures are called as statements. Therefore, PL/SQL procedures are *not* directly callable from SQL statements. However, functions called from a PL/SQL statement or referenced in a SQL expression can call a PL/SQL procedure.

---



---

## Syntax

Use the following syntax to reference a PL/SQL function from SQL:

```
[[schema.]package.]function_name[@dblink][(param_1...param_n)]
```

For example, to reference a function you created that is called `My_func`, in the `My_funcs_pkg` package, in the `Scott` schema, that takes two numeric parameters, you could call the following:

```
SELECT Scott.My_funcs_pkg.My_func(10,20) FROM dual;
```

## Naming Conventions

If only one of the optional schema or package names is given, then the first identifier can be either a schema name or a package name. For example, to determine whether `Payroll` in the reference `Payroll.Tax_rate` is a schema or package name, Oracle proceeds as follows:

- Oracle first checks for the `Payroll` package in the current schema.

- If the `PAYROLL` package is found in the current schema, then Oracle looks for a `Tax_rate` function in the `Payroll` package. If a `Tax_rate` function is not found in the `Payroll` package, then an error message is returned.
- If a `Payroll` package is not found, then Oracle looks for a schema named `Payroll` that contains a top-level `Tax_rate` function. If the `Tax_rate` function is not found in the `Payroll` schema, then an error message is returned.

You can also refer to a stored top-level function using any synonym that you have defined for it.

### Name Precedence

In SQL statements, the names of database columns take precedence over the names of functions with no parameters. For example, if schema `Scott` creates the following two objects:

```
CREATE TABLE Emp_tab(New_sal NUMBER ...);
CREATE FUNCTION New_sal RETURN NUMBER IS ...;
```

Then, in the following two statements, the reference to `New_sal` refers to the column `Emp_tab.New_sal`:

```
SELECT New_sal FROM Emp_tab;
SELECT Emp_tab.New_sal FROM Emp_tab;
```

To access the function `new_sal`, enter the following:

```
SELECT Scott.New_sal FROM Emp_tab;
```

**Example** For example, to call the `Tax_rate` PL/SQL function from schema `Scott`, run it against the `Ss_no` and `sal` columns in `Tax_table`, and place the results in the variable `Income_tax`, specify the following:

---



---

**Note:** You may need to set up data structures similar to the following for certain examples to work:

```
CREATE TABLE Tax_table (
  Ss_no  NUMBER,
  Sal    NUMBER);

CREATE OR REPLACE FUNCTION tax_rate (ssn IN NUMBER, salary IN
NUMBER) RETURN NUMBER IS
  sal_out NUMBER;
BEGIN
  sal_out := salary * 1.1;
END;
```

---



---

```
DECLARE
  Tax_id      NUMBER;
  Income_tax  NUMBER;
BEGIN
  SELECT scott.tax_rate (Ss_no, Sal)
  INTO Income_tax
  FROM Tax_table
  WHERE Ss_no = Tax_id;
END;
```

These sample calls to PL/SQL functions are allowed in SQL expressions:

```
Circle_area(Radius)
Payroll.Tax_rate(Empno)
scott.Payroll.Tax_rate@boston_server(Dependents, Empno)
```

## Arguments

To pass any number of arguments to a function, supply the arguments within the parentheses. You must use positional notation; named notation is not currently supported. For functions that do not accept arguments, use ( ).

## Using Default Values

The stored function `Gross_pay` initializes two of its formal parameters to default values using the `DEFAULT` clause. For example:

```
CREATE OR REPLACE FUNCTION Gross_pay
  (Emp_id  IN NUMBER,
   St_hrs  IN NUMBER DEFAULT 40,
```

```
Ot_hrs IN NUMBER DEFAULT 0) RETURN NUMBER AS
...
```

When calling `Gross_pay` from a procedural statement, you can always accept the default value of `St_hrs`. This is because you can use named notation, which lets you skip parameters. For example:

```
IF Gross_pay(Eenum, Ot_hrs => Otime) > Pay_limit
THEN ...
```

However, when calling `Gross_pay` from a SQL expression, you cannot accept the default value of `St_hrs`, unless you accept the default value of `Ot_hrs`. This is because you cannot use named notation.

### Privileges

To call a PL/SQL function from SQL, you must either own or have `EXECUTE` privileges on the function. To select from a view defined with a PL/SQL function, you must have `SELECT` privileges on the view. No separate `EXECUTE` privileges are necessary to select from the view.

## Meeting Basic Requirements

To be callable from SQL expressions, a user-defined PL/SQL function must meet the following basic requirements:

- It must be a stored function, *not* a function defined within a PL/SQL block or subprogram.
- It must be a row function, *not* a column (group) function; in other words, it cannot take an entire column of data as its argument.
- All its formal parameters must be `IN` parameters; none can be an `OUT` or `IN OUT` parameter.
- The datatypes of its formal parameters must be Oracle Server internal types, such as `CHAR`, `DATE`, or `NUMBER`, *not* PL/SQL types, such as `BOOLEAN`, `RECORD`, or `TABLE`.
- Its return type (the datatype of its result value) must be an Oracle Server internal type.

For example, the following stored function meets the basic requirements:

---



---

**Note:** You may need to set up the following data structures for certain examples to work:

```
CREATE TABLE Payroll(
                Srate                NUMBER
                Orate                NUMBER
                Acctno              NUMBER );
```

---



---

```
CREATE FUNCTION Gross_pay
  (Emp_id IN NUMBER,
   St_hrs IN NUMBER DEFAULT 40,
   Ot_hrs IN NUMBER DEFAULT 0) RETURN NUMBER AS
  St_rate NUMBER;
  Ot_rate NUMBER;

BEGIN
  SELECT Srate, Orate INTO St_rate, Ot_rate FROM Payroll
     WHERE Acctno = Emp_id;
  RETURN St_hrs * St_rate + Ot_hrs * Ot_rate;
END Gross_pay;
```

## Controlling Side Effects

The *purity* of a stored function refers to the side effects of that function on database tables or package variables. Side effects can prevent the parallelization of a query, yield order-dependent (and therefore, indeterminate) results, or require that package state be maintained across user sessions. Various side effects are not allowed when a function is called from a SQL query or DML statement.

In previous releases, Oracle leveraged the PL/SQL compiler to enforce restrictions during the compilation of a stored function or a SQL statement. In Oracle8i, the compile-time restrictions have been relaxed, and a smaller set of restrictions are enforced during execution.

**See Also:** ["Restrictions"](#) on page 9-62.

This change provides uniform support for stored functions written in PL/SQL, Java, and C, and it allows programmers the most flexibility possible.

## PL/SQL Compilation Checking

A user-written function can now be called from a SQL statement *without* any compile-time checking of its purity: `PRAGMA RESTRICT_REFERENCES` is no longer required on functions called from SQL statements.

`PRAGMA RESTRICT_REFERENCES` remains available as a means of asking the PL/SQL compiler to verify that a function has only the side effects that you expect. SQL statements, package variable accesses, or calls to functions that violate the declared restrictions will continue to raise PL/SQL compilation errors to help you isolate the code that has unintended effects.

Because Oracle no longer requires that the pragma on functions called from SQL statements, different applications may choose different style standards on whether and where to use `PRAGMA RESTRICT_REFERENCES`. An existing PL/SQL application will most likely want to continue using the pragma even on new functionality, to ease integration with the existing code. A newly created Java application will most likely not want to use the pragma at all, because the Java compiler does not have the functionality to assist in isolating unintended effects.

**See Also:** ["Using PRAGMA RESTRICT\\_REFERENCES"](#) on page 9-66.

## Restrictions

When a SQL statement is run, checks are made to see if it is logically embedded within the execution of an already running SQL statement. This occurs if the statement is run from a trigger or from a function that was in turn called from the already running SQL statement. In these cases, further checks occur to determine if the new SQL statement is safe in the specific context.

The following restrictions are enforced:

- A function called from a query or DML statement may not end the current transaction, create or rollback to a savepoint, or `ALTER` the system or session.
- A function called from a query (`SELECT`) statement or from a parallelized DML statement may not execute a DML statement or otherwise modify the database.
- A function called from a DML statement may not read or modify the particular table being modified by that DML statement.

These restrictions apply regardless of what mechanism is used to run the SQL statement inside the function or trigger. For example:

- They apply to a SQL statement called from PL/SQL, whether embedded directly in a function or trigger body, run using the new native dynamic mechanism (`EXECUTE IMMEDIATE`), or run using the `DBMS_SQL` package.
- They apply to statements embedded in Java with SQLJ syntax or run using JDBC.
- They apply to statements run with OCI using the callback context from within an "external" C function.

You can avoid these restrictions if the execution of the new SQL statement is not logically embedded in the context of the already running statement. PL/SQL's new autonomous transactions provide one escape. Another escape is available using OCI from an external C function, if you create a new connection, rather than using the handle available from the `OCIExtProcContext` argument.

### Declaring a Function

The keywords `DETERMINISTIC` and `PARALLEL_ENABLE` can be used in the syntax for declaring a function. These are optimization hints, informing the query optimizer and other aspects of Oracle8i about those functions that need not be called redundantly and about those that may be used within a parallelized query or parallelized DML statement. Only functions that are `DETERMINISTIC` are allowed in function-based indexes and in certain snapshots and materialized views.

A function that is dependent solely on the values passed into it as arguments, and does not meaningfully reference or modify the contents of package variables or the database, or have any other side-effects, is termed *deterministic*. Such a function reliably produces the exact same result value for any particular combination of argument values passed into it.

The `DETERMINISTIC` keyword is placed after the return value type in a declaration of the function. For example:

```
CREATE FUNCTION F1 (P1 NUMBER) RETURN NUMBER DETERMINISTIC IS
BEGIN
  RETURN P1 * 2;
END;
```

This keyword may be placed on a function defined in a `CREATE FUNCTION` statement, in a function's declaration in a `CREATE PACKAGE` statement, or on a method's declaration in a `CREATE TYPE` statement. It should not be repeated on the function's or method's body in a `CREATE PACKAGE BODY` or `CREATE TYPE BODY` statement.

Certain performance optimizations occur on calls to functions that are marked `DETERMINISTIC`, without any other action being required. However, the database has no reasonable way to recognize if the function's behavior indeed is truly deterministic. If the `DETERMINISTIC` keyword is applied to a function whose behavior is not truly deterministic, then the result of queries involving that function is unpredictable.

Two new features in Oracle8i require that any function used with them is declared `DETERMINISTIC`.

- Any function used in a function-based index is required to be `DETERMINISTIC`.
- Any function used in a materialized view must be `DETERMINISTIC` if that view is to be marked `ENABLE QUERY REWRITE`.

Both of these features attempt to use previously calculated results rather than calling the function when it is possible to do so.

It is also preferable that only functions declared `DETERMINISTIC` are used in any materialized view or snapshot that is declared `REFRESH FAST`. Oracle allows in `REFRESH FAST` snapshots those functions that have a `PRAGMA RESTRICT_REFERENCES` noting that they are `RNDS`, and those PL/SQL functions defined with a `CREATE FUNCTION` statement whose code can be examined to determine that they do not read the database nor call any other routine which might, as these have been allowed historically.

Functions that are used in a `WHERE`, `ORDER BY`, or `GROUP BY` clause, are `MAP` or `ORDER` methods of a SQL type, or in any other way are part of determining whether or where a row should appear in a result set also should be `DETERMINISTIC` as discussed above. Oracle cannot require that they be explicitly declared `DETERMINISTIC` without breaking existing applications, but the use of the keyword might be a wise choice of style within your application.

### Parallel Query/Parallel DML

Oracle's parallel execution feature divides the work of executing a SQL statement across multiple processes. Functions called from a SQL statement which is run in parallel may have a separate copy run in each of these processes, with each copy called for only the subset of rows that are handled by that process.

Each process has its own copy of package variables. When parallel execution begins, these are initialized based on the information in the package specification and body as if a new user is logging into the system; the values in package variables are not copied from the original login session. And changes made to package variables are not automatically propagated between the various sessions or back to

the original session. Java `STATIC` class attributes are similarly initialized and modified independently in each process. Because a function can use package (or Java `STATIC`) variables to accumulate some value across the various rows it encounters, Oracle cannot assume that it is safe to parallelize the execution of all user-defined functions.

For query (`SELECT`) statements, in previous releases, the parallel query optimization looked to see if a function was noted as `RNPS` and `WNPS` in a `PRAGMA RESTRICT_REFERENCES` declaration; those functions that were marked as both `RNPS` and `WNPS` could be run in parallel. Functions defined with a `CREATE FUNCTION` statement had their code implicitly examined to determine if they were actually pure enough; parallelized execution might occur even though a pragma cannot be specified on these functions.

**See Also:** ["Using PRAGMA RESTRICT\\_REFERENCES"](#) on page 9-66.

For DML statements, in previous releases, the parallelization optimization looked to see if a function was noted as having all four of `RNDS`, `WNDS`, `RNPS` and `WNPS` specified in a `PRAGMA RESTRICT_REFERENCES` declaration; those functions that were marked as neither reading nor writing to either the database or package variables could run in parallel. Again, those functions defined with a `CREATE FUNCTION` statement had their code implicitly examined to determine if they were actually pure enough; parallelized execution might occur even though a pragma cannot be specified on these functions.

In Oracle8i, we continue to parallelize those functions that Oracle7 and Oracle8 would recognize as parallelizable. In addition, a new keyword, `PARALLEL_ENABLE`, has been added. This is the preferred way now for users to mark their code as safe for parallel execution. This keyword is syntactically similar to `DETERMINISTIC` as described above; it is placed after the return value type in a declaration of the function, as in:

```
CREATE FUNCTION F1 (P1 NUMBER) RETURN NUMBER PARALLEL_ENABLE IS
BEGIN
    RETURN P1 * 2;
END;
```

This keyword may be placed on a function defined in a `CREATE FUNCTION` statement, in a function's declaration in a `CREATE PACKAGE` statement, or on a method's declaration in a `CREATE TYPE` statement. It should not be repeated on the function's or method's body in a `CREATE PACKAGE BODY` or `CREATE TYPE BODY` statement.

Note that a PL/SQL function that is defined with `CREATE FUNCTION` may still be run in parallel without any explicit declaration that it is safe to do so, if the system can determine that it neither reads nor writes package variables nor calls any function that might do so. A Java method or C function is never seen by the system as safe to run in parallel unless the programmer explicitly indicates `PARALLEL_ENABLE` on the "call specification" or provides a `PRAGMA RESTRICT_REFERENCES` indicating that the function is sufficiently pure.

An additional runtime restriction is imposed on functions run in parallel as part of a parallelized DML statement. Such a function is not permitted to in turn execute a DML statement; it is subject to the same restrictions that are enforced on functions that are run inside a query (`SELECT`) statement.

**See Also:** ["Restrictions"](#) on page 9-62.

### Using `PRAGMA RESTRICT_REFERENCES`

To assert the purity level, code the pragma `RESTRICT_REFERENCES` in the package specification (not in the package body). The pragma must follow the function declaration, but it does not need to follow it immediately. Only one pragma can reference a given function declaration.

To code the pragma `RESTRICT_REFERENCES`, use the following syntax:

```
PRAGMA RESTRICT_REFERENCES (
    Function_name, WNDS [, WNPS] [, RNDS] [, RNPS] [, TRUST] );
```

Where:

WNDS	Writes no database state (does not modify database tables).
RNDS	Reads no database state (does not query database tables).
WNPS	Writes no package state (does not change the values of packaged variables).
RNPS	Reads no package state (does not reference the values of packaged variables).
TRUST	Allows easy calling from functions that do have <code>RESTRICT_REFERENCES</code> declarations to those that do not.

You can pass the arguments in any order. If any SQL statement inside the function body violates a rule, then you get an error when the statement is parsed.

In the example below, the function `compound` neither reads nor writes database or package state; therefore, you can assert the maximum purity level. Always assert

the highest purity level that a function allows. That way, the PL/SQL compiler never rejects the function unnecessarily.

---

---

**Note:** You may need to set up the following data structures for certain examples to work:

```
CREATE TABLE Accts (  
  Yrs      NUMBER  
  Amt      NUMBER  
  Acctno   NUMBER  
  Rte      NUMBER);
```

---

---

```
CREATE PACKAGE Finance AS -- package specification  
  FUNCTION Compound  
    (Years IN NUMBER,  
     Amount IN NUMBER,  
     Rate IN NUMBER) RETURN NUMBER;  
  PRAGMA RESTRICT_REFERENCES (Compound, WNDS, WNPS, RNDS, RNPS);  
END Finance;
```

```
CREATE PACKAGE BODY Finance AS --package body  
  FUNCTION Compound  
    (Years IN NUMBER,  
     Amount IN NUMBER,  
     Rate IN NUMBER) RETURN NUMBER IS
```

```
BEGIN
    RETURN Amount * POWER((Rate / 100) + 1, Years);
END Compound;
-- no pragma in package body
END Finance;
```

Later, you might call `compound` from a PL/SQL block, as follows:

```
DECLARE
    Interest NUMBER;
    Acct_id NUMBER;
BEGIN
    SELECT Finance.Compound(Yrs, Amt, Rte) -- function call
    INTO   Interest
    FROM   Accounts
    WHERE  Acctno = Acct_id;
```

**Using the Keyword TRUST** The keyword `TRUST` in the `RESTRICT_REFERENCES` syntax allows easy calling from functions that have `RESTRICT_REFERENCES` declarations to those that do not. When `TRUST` is present, the restrictions listed in the pragma are not actually enforced, but rather are simply trusted to be true.

When calling from a section of code that is using pragmas to one that is not, there are two likely usage styles. One is to place a pragma on the routine to be called, for example on a "call specification" for a Java method. Then, calls from PL/SQL to this method will complain if the method is less restricted than the calling function. For example:

```
CREATE OR REPLACE PACKAGE P1 IS
    FUNCTION F1 (P1 NUMBER) RETURN NUMBER IS
        LANGUAGE JAVA NAME 'CLASS1.METHODNAME(int) return int';
        PRAGMA RESTRICT_REFERENCES(F1,WNDS,TRUST);
    FUNCTION F2 (P1 NUMBER) RETURN NUMBER;

    PRAGMA RESTRICT_REFERENCES(F2,WNDS);
END;

CREATE OR REPLACE PACKAGE BODY P1 IS
    FUNCTION F2 (P1 NUMBER) RETURN NUMBER IS
    BEGIN
        RETURN F1(P1);
    END;
END;
```

Here, `F2` can call `F1`, as `F1` has been declared to be `WNDS`.

The other approach is to mark only the caller, which may then make a call to any function without complaint. For example:

```
CREATE OR REPLACE PACKAGE Pl1 IS
  FUNCTION F1 (P1 NUMBER) RETURN NUMBER IS
    LANGUAGE JAVA NAME 'CLASS1.METHODNAME(int) return int';
  FUNCTION F2 (P1 NUMBER) RETURN NUMBER;
  PRAGMA RESTRICT_REFERENCES (F2,WNDS,TRUST);
END;

CREATE OR REPLACE PACKAGE BODY Pl1 IS
  FUNCTION F2 (P1 NUMBER) RETURN NUMBER IS
  BEGIN
    RETURN F1(P1);
  END;
END;
```

Here, F2 can call F1 because while F2 is promised to be WNDS (because TRUST is specified), the body of F2 is not actually examined to determine if it truly satisfies the WNDS restriction. Because F2 is not examined, its call to F1 is allowed, even though there is no PRAGMA RESTRICT\_REFERENCES for F1.

**Differences between Static and Dynamic SQL Statements.** Static INSERT, UPDATE, and DELETE statements do not violate RNDS if these statements do not explicitly read any database states, such as columns of a table. However, dynamic INSERT, UPDATE, and DELETE statements *always* violate RNDS, regardless of whether or not the statements explicitly read database states.

The following INSERT violates RNDS if it's executed dynamically, but it does *not* violate RNDS if it's executed statically.

```
INSERT INTO my_table values(3, 'SCOTT');
```

The following UPDATE always violates RNDS statically and dynamically, because it explicitly reads the column name of my\_table.

```
UPDATE my_table SET id=777 WHERE name='SCOTT';
```

## Overloading

PL/SQL lets you *overload* packaged (but not stand-alone) functions. You can use the same name for different functions if their formal parameters differ in number, order, or datatype family.

However, a `RESTRICT_REFERENCES` pragma can apply to only one function declaration. Therefore, a pragma that references the name of overloaded functions always applies to the nearest foregoing function declaration.

In the following example, the pragma applies to the second declaration of `valid`:

```
CREATE PACKAGE Tests AS
    FUNCTION Valid (x NUMBER) RETURN CHAR;
    FUNCTION Valid (x DATE) RETURN CHAR;
    PRAGMA RESTRICT_REFERENCES (valid, WNDS);
END;
```

## Serially Reusable PL/SQL Packages

PL/SQL packages usually consume user global area (UGA) memory corresponding to the number of package variables and cursors in the package. This limits scalability, because the memory increases linearly with the number of users. The solution is to allow some packages to be marked as `SERIALLY_REUSABLE` (using pragma syntax).

For serially reusable packages, the package global memory is not kept in the UGA per user; rather, it is kept in a small pool and reused for different users. This means that the global memory for such a package is only used within a unit of work. At the end of that unit of work, the memory can therefore be released to the pool to be reused by another user (after running the initialization code for all the global variables).

The unit of work for serially reusable packages is implicitly a call to the server; for example, an OCI call to the server, or a PL/SQL client-to-server RPC call, or a server-to-server RPC call.

### Package States

The state of a nonreusable package (one not marked `SERIALLY_REUSABLE`) persists for the lifetime of a session. A package's *state* includes global variables, cursors, and so on.

The state of a serially reusable package persists only for the lifetime of a call to the server. On a subsequent call to the server, if a reference is made to the serially reusable package, then Oracle creates a new *instantiation* (described below) of the serially reusable package and initializes all the global variables to `NULL` or to the default values provided. Any changes made to the serially reusable package state in the previous calls to the server are not visible.

---

---

**Note:** Creating a new instantiation of a serially reusable package on a call to the server does not necessarily imply that Oracle allocates memory or configures the instantiation object. Oracle simply looks for an available instantiation work area (which is allocated and configured) for this package in a least-recently used (LRU) pool in SGA.

At the end of the call to the server, this work area is returned back to the LRU pool. The reason for keeping the pool in the SGA is that the work area can be reused across users who have requests for the same package.

---

---

### Why Serially Reusable Packages?

Because the state of a non-reusable package persists for the lifetime of the session, this locks up UGA memory for the whole session. In applications, such as Oracle Office, a log-on session can typically exist for days together. Applications often need to use certain packages only for certain localized periods in the session and would ideally like to de-instantiate the package state in the middle of the session, after they are done using the package.

With `SERIALLY_REUSABLE` packages, application developers have a way of modelling their applications to manage their memory better for scalability. Package state that they care about only for the duration of a call to the server should be captured in `SERIALLY_REUSABLE` packages.

### Syntax

A package can be marked serially reusable by a pragma. The syntax of the pragma is:

```
PRAGMA SERIALLY_REUSABLE;
```

A package specification can be marked serially reusable, whether or not it has a corresponding package body. If the package has a body, then the body must have the serially reusable pragma, if its corresponding specification has the pragma; it cannot have the serially reusable pragma unless the specification also has the pragma.

## Semantics

A package that is marked `SERIALLY_REUSABLE` has the following properties:

- Its package variables are meant for use only within the work boundaries, which correspond to calls to the server (either OCI call boundaries or PL/SQL RPC calls to the server).

---

---

**Note:** If the application programmer makes a mistake and depends on a package variable that is set in a previous unit of work, then the application program can fail. PL/SQL cannot check for such cases.

---

---

- A pool of package instantiations is kept, and whenever a "unit of work" needs this package, one of the instantiations is "reused", as follows:
  - The package variables are reinitialized (for example, if the package variables have default values, then those values are reinitialized).
  - The initialization code in the package body is run again.
- At the "end work" boundary, cleanup is done.
  - If any cursors were left open, then they are silently closed.
  - Some non-reusable secondary memory is freed (such as memory for collection variables or long `VARCHAR2`s).
  - This package instantiation is returned back to the pool of reusable instantiations kept for this package.
- Serially reusable packages cannot be accessed from within triggers. If you attempt to access a serially reusable package from a trigger, then Oracle issues the error message "cannot access Serially Reusable package <string> in the context of a trigger."

## Examples

**Example 1** This example has a serially reusable package specification (there is no body). It demonstrates how package variables act across call boundaries.

```
CONNECT Scott/Tiger

CREATE OR REPLACE PACKAGE Sr_pkg IS
  PRAGMA SERIALLY_REUSABLE;
  N NUMBER := 5;           -- default initialization
END Sr_pkg;
```

Suppose your Enterprise Manager (or SQL\*Plus) application issues the following:

```
CONNECT Scott/Tiger

# first CALL to server
BEGIN
  Sr_pkg.N := 10;
END;

# second CALL to server
BEGIN
  DBMS_OUTPUT.PUT_LINE(Sr_pkg.N);
END;
```

The above program prints:

5

---



---

**Note:** If the package had not had the pragma `SERIALLY_REUSABLE`, then the program would have printed '10'.

---



---

**Example 2** This example has both a package specification and package body, which are serially reusable. Like [Example 1](#), this example demonstrates how the package variables act across call boundaries.

```
CONNECT Scott/Tiger

DROP PACKAGE Sr_pkg;
CREATE OR REPLACE PACKAGE Sr_pkg IS
  PRAGMA SERIALLY_REUSABLE;
  TYPE Str_table_type IS TABLE OF VARCHAR2(200) INDEX BY BINARY_INTEGER;
  Num      NUMBER      := 10;
END Sr_pkg;
```

```

Str      VARCHAR2(200) := 'default-init-str';
Str_tab STR_TABLE_TYPE;

PROCEDURE Print_pkg;
PROCEDURE Init_and_print_pkg(N NUMBER, V VARCHAR2);
END Sr_pkg;
CREATE OR REPLACE PACKAGE BODY Sr_pkg IS
  -- the body is required to have the pragma because the
  -- specification of this package has the pragma
  PRAGMA SERIALLY_REUSABLE;
  PROCEDURE Print_pkg IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('num: ' || Sr_pkg.Num);
    DBMS_OUTPUT.PUT_LINE('str: ' || Sr_pkg.Str);
    DBMS_OUTPUT.PUT_LINE('number of table elems: ' || Sr_pkg.Str_tab.Count);
    FOR i IN 1..Sr_pkg.Str_tab.Count LOOP
      DBMS_OUTPUT.PUT_LINE(Sr_pkg.Str_tab(i));
    END LOOP;
  END;
  PROCEDURE Init_and_print_pkg(N NUMBER, V VARCHAR2) IS
  BEGIN
    -- init the package globals
    Sr_pkg.Num := N;
    Sr_pkg.Str := V;
    FOR i IN 1..n LOOP
      Sr_pkg.Str_tab(i) := V || ' ' || i;
    END LOOP;
    -- now print the package
    Print_pkg;
  END;
END Sr_pkg;

SET SERVEROUTPUT ON;

Rem SR package access in a CALL:

BEGIN
  -- initialize and print the package
  DBMS_OUTPUT.PUT_LINE('Initing and printing pkg state..');
  Sr_pkg.Init_and_print_pkg(4, 'abracadabra');
  -- print it in the same call to the server.
  -- we should see the initialized values.
  DBMS_OUTPUT.PUT_LINE('Printing package state in the same CALL...');
  Sr_pkg.Print_pkg;
END;
```

```

Initing and printing pkg state..
num: 4
str: abracadabra
number of table elems: 4
abracadabra 1
abracadabra 2
abracadabra 3
abracadabra 4
Printing package state in the same CALL...
num: 4
str: abracadabra
number of table elems: 4
abracadabra 1
abracadabra 2
abracadabra 3
abracadabra 4

REM SR package access in subsequent CALL:
BEGIN
  -- print the package in the next call to the server.
  -- We should that the package state is reset to the initial (default) values.
  DBMS_OUTPUT.PUT_LINE('Printing package state in the next CALL...');
  Sr_pkg.Print_pkg;
END;
Statement processed.
Printing package state in the next CALL...
num: 10
str: default-init-str
number of table elems: 0

```

**Example 3** This example demonstrates that any open cursors in serially reusable packages get closed automatically at the end of a work boundary (which is a call). *Also, in a new call, these cursors need to be opened again.*

```

REM For serially reusable pkg: At the end work boundaries
REM (which is currently the OCI call boundary) all open
REM cursors will be closed.
REM
REM Because the cursor is closed - every time we fetch we
REM will start at the first row again.

```

```

CONNECT Scott/Tiger
DROP PACKAGE Sr_pkg;
DROP TABLE People;

```

```
CREATE TABLE People (Name VARCHAR2(20));
INSERT INTO People VALUES ('ET');
INSERT INTO People VALUES ('RAMBO');
CREATE OR REPLACE PACKAGE Sr_pkg IS
    PRAGMA SERIALLY_REUSABLE;
    CURSOR C IS SELECT Name FROM People;
END Sr_pkg;
SQL> SET SERVEROUTPUT ON;
SQL>
CREATE OR REPLACE PROCEDURE Fetch_from_cursor IS
Name VARCHAR2(200);
BEGIN
    IF (Sr_pkg.C%ISOPEN) THEN
        DBMS_OUTPUT.PUT_LINE('cursor is already open.');
```

ELSE

```
        DBMS_OUTPUT.PUT_LINE('cursor is closed; opening now.');
```

OPEN Sr\_pkg.C;

```
END IF;
-- fetching from cursor.
FETCH sr_pkg.C INTO name;
DBMS_OUTPUT.PUT_LINE('fetched: ' || Name);
FETCH Sr_pkg.C INTO name;
DBMS_OUTPUT.PUT_LINE('fetched: ' || Name);
-- Oops forgot to close the cursor (Sr_pkg.C).
-- But, because it is a Serially Reusable pkg's cursor,
-- it will be closed at the end of this CALL to the server.
END;
EXECUTE fetch_from_cursor;
cursor is closed; opening now.
fetched: ET
fetched: RAMBO
```

### The Need to Work with Multiple Languages

Oracle offers you the possibility of working in different languages:

- **PL/SQL**, as described in the *PL/SQL User's Guide and Reference*
- **C**, by means of the Oracle Call Interface (OCI), as described in the *Oracle Call Interface Programmer's Guide*
- **C++**, by means of the Pro\*C/C++ precompiler, as described in the *Pro\*C/C++ Precompiler Programmer's Guide*
- **COBOL**, by means of the Pro\*COBOL precompiler, as described in the *Pro\*COBOL Precompiler Programmer's Guide*
- **Visual Basic**, by means of Oracle Objects For OLE (OO4O), as described in the *Oracle Objects for OLE/ActiveX Programmer's Guide*
- **Java**, by means of the JDBC Application Programmers Interface (API), as described in *Oracle8 Database Programming with Java*.

How should you choose between these different implementation possibilities? Each of these languages offers different advantages: Ease of use, the availability of programmers with specific expertise, the need for portability, and the existence of legacy code are powerful determinants.

However, the choice may narrow depending on how your application needs to work with the Oracle ORDBMS:

- PL/SQL is a powerful development tool, specialized for SQL transaction processing.
- Some computation-intensive tasks are executed most efficiently in a lower level language, such as C.

- The need for portability, together with the need for security, may influence you to select Java.

Most significantly, from the point of view of performance, you should note that only PL/SQL and Java methods run within the address space of the server. C/C++ methods are dispatched as external procedures, and run on the server but outside the address space of the server. Pro\*COBOL and Pro\*C are precompilers, and Visual Basic accesses Oracle via the OCI, which is implemented in C.

Taking all these factors into account suggests that there may be a number of situations in which you may need to implement your application in more than one language. For instance, the introduction of Java running within the address space of the server suggest that you may want to import existing Java applications into the database, and then leverage this technology by calling Java functions from PL/SQL and SQL.

Until Oracle 8.0, the Oracle RDBMS supported SQL and the stored procedure language PL/SQL. In Oracle 8.0, PL/SQL introduced **external procedures**, which allowed the capability of writing C functions as PL/SQL bodies. These C functions are callable from PL/SQL and SQL (via PL/SQL). With 8.1, Oracle provides a special-purpose interface, the **call specification**, that lets you call **external routines** from other languages. While this service is designed for intercommunication between SQL, PL/SQL, C, and Java, it is accessible from any base language that can call these languages. For example, your routine can be written in a language other than Java or C and still be usable by SQL or PL/SQL, provided that is callable by C. Therefore, if you have a candidate C++ routine, you would use a C++ extern "C" statement in that routine to make it callable by C.

This means that the strengths and capabilities of different languages are available to you, irrespective of your programmatic environment: You are not restricted to one language with its inherent limitations. The use of external routines promotes reusability and modularity because you can deploy specific languages for specific purposes.

## What is an External Routine?

An *external routine*, previously referred to as an *external procedure*, is a routine stored in a **dynamic link library** (DLL), or **libunit** in the case of a Java class method. You register the routine with the base language, and then call it to perform special-purpose processing.

For instance, if you are working in PL/SQL, then the language loads the library dynamically at runtime, and then calls the routine as if it were a PL/SQL

subprogram. These routines participate fully in the current transaction and can 'call back' to the database to perform SQL operations.

The routines are loaded only when necessary, so memory is conserved. The decoupling of the call specification from its implementation body means that the routines can be enhanced without affecting the calling programs.

External routines let you:

- Move computation-bound programs from client to server where they execute faster (because they avoid the roundtrips entailed in across-network communication)
- Interface the database server with external systems and data sources
- Extend the functionality of the database server itself

## The Call Specification

Until now, you published an external routine to Oracle via an `AS EXTERNAL` clause in a PL/SQL wrapper. This wrapper defined the mapping to, and allowed the calling of, external C routines. Oracle 8.1 introduces **call specifications**, which include the `AS EXTERNAL` wrapper as a subset of the new `AS LANGUAGE` clause. `AS LANGUAGE` call specifications allow the publishing of external C routines, as before, but also Java class methods.

---

---

**Note:** Call specifications also allow you to publish with the `AS EXTERNAL` clause, introduced in Oracle 8.0. For new applications, however, you should use the `AS LANGUAGE` clause.

---

---

In general, call specifications enable:

- Dispatching the appropriate C or Java target routine
- Datatype conversions
- Parameter mode mappings
- Automatic memory allocation and cleanup
- Purity constraints to be specified, where necessary, for packaged functions called from SQL.
- Calling Java methods or C routines from database triggers

- **Location flexibility:** You can put `AS LANGUAGE` call specifications in package or type specifications, or package (or type) bodies to optimize performance and hide implementation details

To use an already-existing program as an external routine, load, publish, and then call it.

## Loading External Routines

To make your external C routines or Java methods available to PL/SQL, you must first load them. The manner of doing this depends upon whether the routine is written in C or Java.

**See Also:** *Oracle8i Java Stored Procedures Developer's Guide*

For help in creating a DLL, look in the RDBMS subdirectory `/public`, where a template *makefile* can be found.

## Loading Java Class Methods

One way to load Java programs is to use the `CREATE JAVA` statement, which you can execute interactively from SQL\*Plus. When implicitly invoked by the `CREATE JAVA` statement, the Java Virtual Machine (JVM) library manager loads Java binaries (`.class` files) and resources from local `BFILES` or `LOB` columns into RDBMS libunits.

Suppose a compiled Java class is stored in the following OS file:

```
/home/java/bin/Agent.class
```

Creating a class libunit in schema `scott` from file `Agent.class` requires two steps: First, create a directory object on the server's file system. The name of the directory object is an alias for the directory path leading to `Agent.class`.

To create the directory object, you must grant user `scott` the `CREATE ANY DIRECTORY` privilege, then execute the `CREATE DIRECTORY` statement, as follows:

```
CONNECT System/Manager
GRANT CREATE ANY DIRECTORY TO Scott IDENTIFIED BY Tiger;
CONNECT Scott/Tiger
CREATE DIRECTORY Bfile_dir AS '/home/java/bin';
```

Now, you are ready to create the class libunit, as follows:

```
CREATE JAVA CLASS USING BFILE (Bfile_dir, 'Agent.class');
```

The name of the libunit is derived from the name of the class.

Alternatively, you can use the command-line utility `LoadJava`. This uploads Java binaries and resources into a system-generated database table, then uses the `CREATE JAVA` statement to load the Java files into RDBMS libunits. You can upload Java files from OS file systems, Java IDEs, intranets, or the Internet.

**See Also:** *Oracle8i Java Stored Procedures Developer's Guide*

## Loading External C Routines

In order to set up to use external routines written in C, or callable by C, you and your DBA take the following steps:

---

---

**Note:** This feature is available only on platforms that support DLLs or dynamically loadable shared libraries such as Solaris `.so` libraries.

---

---

### 1. Set Up the Environment

Your DBA sets up the environment for calling external routines by adding entries to the files `tnsname.ora` and `listener.ora` and by starting a Listener process exclusively for external routines.

**See Also:** *Oracle8i Administrator's Guide*.

The Listener sets a few required environment variables (such as `ORACLE_HOME`, `ORACLE_SID`, and `LD_LIBRARY_PATH`) for `extproc`. Otherwise, it provides `extproc` with a "clean" environment. The environment variables set for `extproc` are independent of those set for the client, server, and Listener. Therefore, external routines, which run in the `extproc` process, cannot read environment variables set for the client, server, or Listener process.

---

---

**Note:** It is possible for you to set and read environment variables themselves by using the standard C routines `setenv()` and `getenv()`, respectively. Environment variables, set this way, are specific to the `extproc` process, which means that they can be read by all functions executed in that process, but not by any other process running on the same machine.

---

---

## 2. Identify the DLL

In this context, a DLL is any dynamically loadable operating-system file that stores external routines.

For safety, your DBA controls access to the DLL. Using the `CREATE LIBRARY` statement, the DBA creates a schema object called an **alias library**, which represents the DLL. Then, if you are an authorized user, the DBA grants you `EXECUTE` privileges on the alias library. Alternatively, the DBA may you `CREATE ANY LIBRARY` privileges, in which case you can create your own alias libraries using the following syntax:

```
CREATE LIBRARY library_name {IS | AS} 'file_path';
```

You must specify the full path to the DLL, because the linker cannot resolve references to just the DLL name. In the following example, you create alias library `c_utils`, which represents DLL `utils.so`:

```
CREATE LIBRARY C_utils AS '/DLLs/utils.so';
```

## 3. Designate the External Routine

You find or write a new external C routine, then add it to the DLL, or simply designate a routine already in the DLL.

External C routines are loaded into DLLs. After creating and including your external routine within a DLL, you create the alias library which represents the DLL, like this:

```
CREATE LIBRARY C_utils AS '/DLLs/utils.so';
```

# Publishing an External Routine

Oracle can only use external routines that have been published. Publishing is accomplished with a call specification, which maps names, parameter types, and return types for your Java class method or C external routine to their SQL

counterparts. It is written like any other PL/SQL stored subprogram except that, in its body, instead of declarations and a BEGIN.. END block, you code the AS LANGUAGE clause.

The call specification syntax, which follows the normal CREATE OR REPLACE syntax for a procedure, function, package specification, package body, type specification, or type body, is:

```
{IS | AS} LANGUAGE {C | JAVA}
```

---



---

**Note:** Oracle uses a PL/SQL variant of the ANSI SQL92 External Procedure, but replaces the ANSI keyword AS EXTERNAL with this call specification syntax. This new syntax, introduced for Java class methods, has now been extended to C routines.

*Oracle8i Java Stored Procedures Developer's Guide*

---



---

This is then followed by either:

```
NAME <java_string_literal_name>
```

Where *java\_string\_literal\_name* is the signature of your Java method, or by:

```
LIBRARY <library_name>
[NAME <c_string_literal_name>]
[WITH CONTEXT]
[PARAMETERS (external_parameter[, external_parameter]...)];
```

Where *library\_name* is the name of your alias library, *c\_string\_literal\_name* is the name of your external C routine, and *external\_parameter* stands for:

```
{ CONTEXT
  | SELF [{TDO | property}]
  | {parameter_name | RETURN} [property] [BY REFERENCE] [external_datatype]}
```

*property* stands for:

```
{INDICATOR [{STRUCT | TDO}] | LENGTH | MAXLEN | CHARSETID | CHARSETFORM}
```

---

---

**Note:** Unlike Java, C doesn't understand SQL types; therefore, the syntax is more intricate

---

---

## The AS LANGUAGE Clause for Java Class Methods

The [AS] LANGUAGE clause is the interface between PL/SQL and a Java class method.

**See Also:** *Oracle8i Java Stored Procedures Developer's Guide*

## The AS LANGUAGE Clause for External C Routines

The following subclauses tell PL/SQL where to locate the external C routine, how to call it, and what to pass to it. Only the LIBRARY subclause is required.

### LIBRARY

Specifies a local alias library. (You cannot use a database link to specify a remote library.) The library name is a PL/SQL identifier. Therefore, if you enclose the name in double quotes, then it becomes case sensitive. (By default, the name is stored in upper case.) You must have EXECUTE privileges on the alias library.

### NAME

Specifies the external C routine to be called. If you enclose the routine name in double quotes, then it becomes case sensitive. (By default, the name is stored in upper case.) If you omit this subclause, then the routine name defaults to the upper-case name of the PL/SQL subprogram.

---

---

**Note:** The terms LANGUAGE and CALLING STANDARD apply only to the superseded AS EXTERNAL clause.

---

---

### LANGUAGE

Specifies the third-generation language in which the external routine was written. If you omit this subclause, then the language name defaults to C.

### CALLING STANDARD

Specifies the Windows NT calling standard (C or Pascal) under which the external routine was compiled. (Under the Pascal Calling Standard, arguments are reversed

on the stack, and the called function must pop the stack.) If you omit this subclause, then the calling standard defaults to C.

### WITH CONTEXT

Specifies that a context pointer will be passed to the external routine. The context data structure is opaque to the external routine but is available to service routines called by the external routine.

### PARAMETERS

Specifies the positions and datatypes of parameters passed to the external routine. It can also specify parameter properties, such as current length and maximum length, and the preferred parameter passing method (by value or by reference).

## Publishing Java Class Methods

Java classes and their methods are stored in RDBMS libunits in which you can load Java sources, binaries and resources using the `LOADJAVA` utility or the `CREATEJAVA` SQL statements. Libunits can be considered analogous to DLLs written, for example, in C—although they map one-to-one with Java classes, whereas DLLs can contain more than one routine.

**See Also:** *Oracle8i Java Stored Procedures Developer's Guide*

The `NAME`-clause string uniquely identifies the Java method. The PL/SQL function or procedure and Java must correspond with regard to parameters. If the Java method takes no parameters, then you must code an empty parameter list for it.

When you load Java classes into the RDBMS, they are not published to SQL automatically. This is because the methods of many Java classes are called only from other Java classes, or take parameters for which there is no appropriate SQL type.

Suppose you want to publish the following Java method named `J_calcFactorial`, which returns the factorial of its argument:

```
package myRoutines.math;
public class Factorial {
    public static int J_calcFactorial (int n) {
        if (n == 1) return 1;
        else return n * J_calcFactorial(n - 1);
    }
}
```

The following call specification publishes Java method `J_calcFactorial` as PL/SQL stored function `plsToJavaFac_func`, using SQL\*Plus:

```
CREATE OR REPLACE FUNCTION Plstojavafac_func (N NUMBER) RETURN NUMBER AS
  LANGUAGE JAVA
  NAME 'myRoutines.math.Factorial.J_calcFactorial(int) return int';
```

## Publishing External C Routines

In the following example, you write a PL/SQL standalone function named `plsCallsCdivisor_func` that publishes C function `Cdivisor_func` as an external function:

```
CREATE OR REPLACE FUNCTION Plscallsdivisor_func (
  /* Find greatest common divisor of x and y: */
  x    BINARY_INTEGER,
  y    BINARY_INTEGER)
RETURN BINARY_INTEGER
AS LANGUAGE C
  LIBRARY C_utils
  NAME "Cdivisor_func"; /* Quotation marks preserve case. */
```

## Locations of Call Specifications

For both Java class methods and external C routines, call specifications can be specified in any of the following locations:

- Standalone PL/SQL Procedures and Functions
- PL/SQL Package Specifications
- PL/SQL Package Bodies
- Object Type Specifications
- Object Type Bodies

---

---

**Note:** Under Oracle 8.0, `AS EXTERNAL` call specifications could not be placed in package or type bodies.

---

---

We have already shown an example of call specification located in a standalone PL/SQL function. Here are some examples showing some of the other locations.

---

---

**Note:** In the following examples, the `AUTHID` and `SQL_NAME_RESOLVE` clauses may or may not be required to fully stipulate a call specification. See the *Invoker-rights* section of this manual for rules on their placement and defaults.

---

---

### Example: Locating a Call Specification in a PL/SQL Package

```
CREATE OR REPLACE PACKAGE Demo_pack
AUTHID DEFINER
AS
    PROCEDURE plsToC_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
AS LANGUAGE C
    NAME "C_demoExternal"
    LIBRARY SomeLib
    WITH CONTEXT
    PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE);
END;
```

### Example: Locating a Call Specification in a PL/SQL Package Body

```
CREATE OR REPLACE PACKAGE Demo_pack
    AUTHID CURRENT_USER
AS
    PROCEDURE plsToC_demoExternal_proc(x BINARY_INTEGER, y VARCHAR2, z DATE);
END;

CREATE OR REPLACE PACKAGE BODY Demo_pack
    SQL_NAME_RESOLVE CURRENT_USER
AS
    PROCEDURE plsToC_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
AS LANGUAGE JAVA
    NAME 'pkg1.class4.methodProc1(int, java.lang.String, java.sql.Date)';
END;
```

## Example: Locating a Call Specification in an Object Type Specification

**Note:** You may need to set up the following data structures for certain examples to work:

```
CONN SYS/CHANGE_ON_INSTALL AS SYSDBA;
GRANT CREATE ANY LIBRARY TO scott;
CONNECT scott/tiger
CREATE OR REPLACE LIBRARY SOME LIB AS '/tmp/lib.so';
```

```
CREATE OR REPLACE TYPE Demo_typ
AUTHID DEFINER
AS OBJECT
  (Attribute1 VARCHAR2(2000), SomeLib varchar2(20),
  MEMBER PROCEDURE plsToC_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2, z
DATE)
  AS LANGUAGE C
  NAME "C_demoExternal"
  LIBRARY SomeLib
  WITH CONTEXT
  -- PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE)
  PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE, SELF)
);
```

## Example: Locating a Call Specification in an Object Type Body

```
CREATE OR REPLACE TYPE Demo_typ
AUTHID CURRENT_USER
AS OBJECT
  (attribute1 NUMBER,
  MEMBER PROCEDURE plsToJ_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2, z
DATE)
  );

CREATE OR REPLACE TYPE BODY Demo_typ
AS
  MEMBER PROCEDURE plsToJ_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2, z
DATE)
  AS LANGUAGE JAVA
  NAME 'pkg1.class4.J_demoExternal(int, java.lang.String, java.sql.Date)';
END;
```

**Example: Java with AUTHID**

Here is an example of a publishing a Java class method in a standalone PL/SQL subprogram.

```
CREATE OR REPLACE PROCEDURE plsToJ_demoExternal_proc (x BINARY_INTEGER, y
VARCHAR2, z DATE)
    AUTHID CURRENT_USER
AS LANGUAGE JAVA
    NAME 'pkg1.class4.methodProc1(int,java.lang.String,java.sql.Date)';
```

**Example: C with Optional AUTHID**

Here is an example of AS EXTERNAL publishing a C routine in a standalone PL/SQL program, in which the AUTHID clause is optional. This maintains compatibility with the external procedures of Oracle 8.0.

```
CREATE OR REPLACE PROCEDURE plsToC_demoExternal_proc (x BINARY_INTEGER, y
VARCHAR2, z DATE)
AS
    EXTERNAL
    LANGUAGE C
    NAME "C_demoExternal"
    LIBRARY SomeLib
    WITH CONTEXT
    PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE);
```

**Example: Mixing Call Specifications in a Package**

```
CREATE OR REPLACE PACKAGE Demo_pack
AUTHID DEFINER
AS
    PROCEDURE plsToC_InBodyOld_proc (x BINARY_INTEGER, y VARCHAR2, z DATE);
    PROCEDURE plsToC_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2, z DATE);
    PROCEDURE plsToC_InBody_proc (x BINARY_INTEGER, y VARCHAR2, z DATE);
    PROCEDURE plsToJ_InBody_proc (x BINARY_INTEGER, y VARCHAR2, z DATE);

    PROCEDURE plsToJ_InSpec_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
    IS LANGUAGE JAVA
        NAME 'pkg1.class4.J_InSpec_meth(int,java.lang.String,java.sql.Date)';

    PROCEDURE C_InSpec_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
    AS LANGUAGE C
        NAME "C_demoExternal"
        LIBRARY SomeLib
        WITH CONTEXT
        PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE);
```

```
END;

CREATE OR REPLACE PACKAGE BODY Demo_pack
AS
PROCEDURE plsToC_InBodyOld_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
  AS EXTERNAL
  LANGUAGE C
  NAME "C_InBodyOld"
  LIBRARY SomeLib
  WITH CONTEXT
  PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE);
PROCEDURE plsToC_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
  AS LANGUAGE C
  NAME "C_demoExternal"
  LIBRARY SomeLib
  WITH CONTEXT
  PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE);

PROCEDURE plsToC_InBody_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
  AS LANGUAGE C
  NAME "C_InBody"
  LIBRARY SomeLib
  WITH CONTEXT
  PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE);
PROCEDURE plsToJ_InBody_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
  IS LANGUAGE JAVA
  NAME 'pkg1.class4.J_InBody_meth(int, java.lang.String, java.sql.Date)';
END;
```

## Passing Parameters to Java Class Methods with Call Specifications

**See Also:** *Oracle8i Java Stored Procedures Developer's Guide*

## Passing Parameters to External C Routines with Call Specifications

Call specifications allows a mapping between PL/SQL and C datatypes. Datatype mappings are shown below.

Passing parameters to an external C routine is complicated by several circumstances:

- The available set of PL/SQL datatypes does not correspond one-to-one with the set of C datatypes.

- Unlike C, PL/SQL includes the RDBMS concept of nullity. Therefore, PL/SQL parameters can be `NULL`, whereas C parameters cannot.
- The external routine might need the current length or maximum length of `CHAR`, `LONG RAW`, `RAW`, and `VARCHAR2` parameters.
- The external routine might need character set information about `CHAR`, `VARCHAR2`, and `CLOB` parameters.
- PL/SQL might need the current length, maximum length, or null status of values returned by the external routine.

In the following sections, you learn how to specify a parameter list that deals with these circumstances.

---

---

**Note:** The maximum number of parameters that you can pass to a C external routine is 128. However, if you pass float or double parameters by value, then the maximum is less than 128. How much less depends on the number of such parameters and your operating system. To get a rough estimate, count each float or double passed by value as two parameters.

---

---

## Specifying Datatypes

Do not pass parameters to an external routine directly. Instead, pass them to the PL/SQL subprogram that published the external routine. Therefore, you must specify PL/SQL datatypes for the parameters. Each PL/SQL datatype maps to a default external datatype, as shown in [Table 10–1](#).

**Table 10–1 Parameter Datatype Mappings**

PL/SQL Type	Supported External Types	Default External Type
BINARY_INTEGER	[UNSIGNED] CHAR	INT
BOOLEAN	[UNSIGNED] SHORT	
PLS_INTEGER	[UNSIGNED] INT	
	[UNSIGNED] LONG	
	SB1, SB2, SB4	
	UB1, UB2, UB4	
	SIZE_T	
NATURAL	[UNSIGNED] CHAR	UNSIGNED INT
NATURALN	[UNSIGNED] SHORT	
POSITIVE	[UNSIGNED] INT	
POSITIVEN	[UNSIGNED] LONG	
SIGNTYPE	SB1, SB2, SB4	
	UB1, UB2, UB4	
	SIZE_T	
FLOAT	FLOAT	FLOAT
REAL		
DOUBLE PRECISION	DOUBLE	DOUBLE
CHAR	STRING	STRING
CHARACTER	OCISTRING	
LONG		
NCHAR		
NVARCHAR2		
ROWID		
VARCHAR		
VARCHAR2		
LONG RAW	RAW	RAW
RAW	OCIRAW	
BFILE	OCILOBLOCATOR	OCILOBLOCATOR
BLOB		
CLOB		
NCLOB		

**Table 10–1 Parameter Datatype Mappings (Cont.)**

NUMBER	[UNSIGNED] CHAR	OCINUMBER
DEC	[UNSIGNED] SHORT	
DECIMAL	[UNSIGNED] INT	
INT	[UNSIGNED] LONG	
INTEGER	SB1, SB2, SB4	
NUMERIC	UB1, UB2, UB4	
SMALLINT	SIZE_T	
	OCINUMBER	
DATE	OCIDATE	OCIDATE
composite object types: ADTs	dvoid	dvoid
composite object types: collections (varrays, nested tables, index-by tables	OCICOLL	OCICOLL

## External Datatype Mappings

Each external datatype maps to a C datatype, and the datatype conversions are performed implicitly. To avoid errors when declaring C prototype parameters, refer to [Table 10-2](#), which shows the C datatype to specify for a given external datatype and PL/SQL parameter mode. For example, if the external datatype of an OUT parameter is STRING, then specify the datatype `char *` in your C prototype.

**Table 10-2 External Datatype Mappings**

External Datatype	Datatypes Used in C Prototype		
	IN, RETURN	IN by REFERENCE, RETURN by REFERENCE	IN OUT, OUT
CHAR	char	char *	char *
UNSIGNED CHAR	unsigned char	unsigned char *	unsigned char *
SHORT	short	short *	short *
UNSIGNED SHORT	unsigned short	unsigned short *	unsigned short *
INT	int	int *	int *
UNSIGNED INT	unsigned int	unsigned int *	unsigned int *
LONG	long	long *	long *
UNSIGNED LONG	unsigned long	unsigned long *	unsigned long *
SIZE_T	size_t	size_t *	size_t *
SB1	sb1	sb1 *	sb1 *
UB1	ub1	ub1 *	ub1 *
SB2	sb2	sb2 *	sb2 *
UB2	ub2	ub2 *	ub2 *
SB4	sb4	sb4 *	sb4 *
UB4	ub4	ub4 *	ub4 *
FLOAT	float	float *	float *
DOUBLE	double	double *	double *
STRING	char *	char *	char *
RAW	unsigned char *	unsigned char *	unsigned char *
OCILOBLOCATOR	OCILOBLocator *	OCILOBLocator **	OCILOBLocator **

**Table 10–2 (Cont.) External Datatype Mappings**

OCINUMBER	OCINumber *	OCINumber *	OCINumber *
OCISTRING	OCIString *	OCIString *	OCIString *
OCIRAW	OCIRaw *	OCIRaw *	OCIRaw *
OCIDATE	OCIDate *	OCIDate *	OCIDate *
OCICOLL	OCIColl * or OCIArray *, or OCITable *	OCIColl ** or OCIArray **, or OCITable **	OCIColl ** or OCIArray **, or OCITable **
OCITYPE	OCIType *	OCIType *	OCIType *
TDO	OCIType *	OCIType *	OCIType *
ADT	dvoid*	dvoid*	dvoid*

Composite object types are not self describing. Their description is stored in a **Type Descriptor Object (TDO)**. Objects and indicator structs for objects have no predefined OCI datatype, but must use the datatypes generated by Oracle's **Object Type Translator (OTT)**. The optional TDO argument for `INDICATOR`, and for composite objects, in general, has the C datatype, `OCIType *`.

`OCICOLL` for `REF` and collection arguments *is* optional and only exists for the sake of completeness. You can not map `REFs` or collections onto any other datatype and vice versa.

## BY VALUE/REFERENCE for IN and IN OUT Parameter Modes

If you specify `BY VALUE`, then scalar `IN` and `RETURN` arguments are passed by value (which is also the default). Alternatively, you may have them passed by reference by specifying `BY REFERENCE`.

By default, or if you specify `BY REFERENCE`, then scalar `IN OUT`, and `OUT` arguments are passed by reference. Specifying `BY VALUE` for `IN OUT`, and `OUT` arguments is not supported for C. The usefulness of the `BY REFERENCE/VALUE` clause is restricted to external datatypes that are, by default, passed by value. This is true for `IN`, and `RETURN` arguments of the following external types:

```
[UNSIGNED] CHAR
[UNSIGNED] SHORT
[UNSIGNED] INT
[UNSIGNED] LONG
SIZE_T
```

SB1  
SB2  
SB4  
UB1  
UB2  
UB4  
FLOAT  
DOUBLE

All IN and RETURN arguments of external types not on the above list, all IN OUT arguments, and all OUT arguments are passed by reference.

## The PARAMETERS Clause

Generally, the PL/SQL subprogram that publishes an external routine declares a list of formal parameters, as the following example shows:

---

---

**Note:** You may need to set up the following data structures for certain examples to work:

```
CREATE LIBRARY MathLib AS '/tmp/math.so';
```

---

---

```
CREATE OR REPLACE FUNCTION Interp_func (  
  /* Find the value of y at x degrees using Lagrange interpolation: */  
  x    IN FLOAT,  
  y    IN FLOAT)  
RETURN FLOAT AS  
  LANGUAGE C  
  NAME "Interp_func"  
  LIBRARY MathLib;
```

Each formal parameter declaration specifies a name, parameter mode, and PL/SQL datatype (which maps to the default external datatype). That might be all the information the external routine needs. If not, then you can provide more information using the PARAMETERS clause, which lets you specify the following:

- Non-default external datatypes
- The current and/or maximum length of a parameter
- NULL/NOT NULL indicators for parameters
- Characterset IDs and forms
- The position of parameters in the list

- How `IN` parameters are passed (by value or by reference)

If you decide to use the `PARAMETERS` clause, keep in mind:

- For every formal parameter, there must be a corresponding parameter in the `PARAMETERS` clause.
- If you include the `WITH CONTEXT` clause, then you must specify the parameter `CONTEXT`, which shows the position of the context pointer in the parameter list.
- If the external routine is a function, then you must specify the parameter `RETURN`, and it must be in the last position.

## Overriding Default Datatype Mapping

In some cases, you can use the `PARAMETERS` clause to override the default datatype mappings. For example, you can re-map the PL/SQL datatype `BOOLEAN` from external datatype `INT` to external datatype `CHAR`.

## Specifying Properties

You can also use the `PARAMETERS` clause to pass additional information about PL/SQL formal parameters and function results to an external routine. Do this by specifying one or more of the following properties:

```
INDICATOR [ {STRUCT | TDO} ]  
LENGTH  
MAXLEN  
CHARSETID  
CHARSETFORM  
SELF
```

The following table shows the allowed and the default external datatypes, PL/SQL datatypes, and PL/SQL parameter modes allowed for a given property. Notice that `MAXLEN` (used to specify data returned from C back to PL/SQL) cannot be applied to an `IN` parameter.

**Table 10–3 Property Datatype Mappings**

Property	C Parameter		PL/SQL Parameter		
	Allowed External Types	Default External Type	Allowed Types	Allowed Modes	Default Passing Method
INDICATOR	SHORT	SHORT	all scalars	IN	BY VALUE
	INT			IN OUT	BY REFERENCE
	LONG			OUT	BY REFERENCE
				RETURN	BY REFERENCE
LENGTH	[UNSIGNED] SHORT	INT	CHAR	IN	BY VALUE
	[UNSIGNED] INT		LONG RAW	IN OUT	BY REFERENCE
	[UNSIGNED] LONG		RAW	OUT	BY REFERENCE
			VARCHAR2	RETURN	BY REFERENCE
MAXLEN	[UNSIGNED] SHORT	INT	CHAR	IN OUT	BY REFERENCE
	[UNSIGNED] INT		LONG RAW	OUT	BY REFERENCE
	[UNSIGNED] LONG		RAW	RETURN	BY REFERENCE
			VARCHAR2		
CHARSETID	[UNSIGNED] SHORT	[UNSIGNED] INT	CHAR	IN	BY VALUE
CHARSETFORM	[UNSIGNED] INT		CLOB	IN OUT	BY REFERENCE
	[UNSIGNED] LONG		VARCHAR2	OUT	BY REFERENCE
			RETURN	BY REFERENCE	

In the following example, the `PARAMETERS` clause specifies properties for the PL/SQL formal parameters and function result:

```
CREATE OR REPLACE FUNCTION plsToCparse_func (
    x    IN BINARY_INTEGER,
    y    IN OUT CHAR)
RETURN CHAR AS LANGUAGE C
    LIBRARY c_utils
    NAME "C_parse"
    PARAMETERS (
        x,           -- stores value of x
        x INDICATOR, -- stores null status of x
        y,           -- stores value of y
        y LENGTH,   -- stores current length of y
        y MAXLEN,   -- stores maximum length of y
        RETURN INDICATOR,
        RETURN);
```

With this `PARAMETERS` clause, the C prototype becomes:

```
char * C_parse(int x, short x_ind, char *y, int *y_len,  
              int *y_maxlen, short *retind);
```

The additional parameters in the C prototype correspond to the `INDICATOR` (for `x`), `LENGTH` (of `y`), and `MAXLEN` (of `y`], as well as the `INDICATOR` for the function result in the `PARAMETERS` clause. The parameter `RETURN` corresponds to the C function identifier, which stores the result value.

## INDICATOR

An `INDICATOR` is a parameter whose value indicates whether or not another parameter is `NULL`. PL/SQL does not need indicators, because the RDBMS concept of nullity is built into the language. However, an external routine might need to know if a parameter or function result is `NULL`. Also, an external routine might need to signal the server that a returned value is actually a `NULL`, and should be treated accordingly.

In such cases, you can use the property `INDICATOR` to associate an indicator with a formal parameter. If the PL/SQL subprogram is a function, then you can also associate an indicator with the function result, as shown above.

To check the value of an indicator, you can use the constants `OCI_IND_NULL` and `OCI_IND_NOTNULL`. If the indicator equals `OCI_IND_NULL`, then the associated parameter or function result is `NULL`. If the indicator equals `OCI_IND_NOTNULL`, then the parameter or function result is not `NULL`.

For `IN` parameters, which are inherently read-only, `INDICATOR` is passed by value (unless you specify `BY REFERENCE`) and is read-only (even if you specify `BY REFERENCE`). For `OUT`, `IN OUT`, and `RETURN` parameters, `INDICATOR` is passed by reference by default.

The `INDICATOR` can also have a `STRUCT` or `TDO` option. Because specifying `INDICATOR` as a property of an object is not supported, and because arguments of objects have complete indicator structs instead of `INDICATOR` scalars, you must specify this by using the `STRUCT` option. You must use the type descriptor object (`TDO`) option for composite objects and collections,

## LENGTH and MAXLEN

In PL/SQL, there is no standard way to indicate the length of a `RAW` or string parameter. However, in many cases, you want to pass the length of such a parameter to and from an external routine. Using the properties `LENGTH` and `MAXLEN`, you can specify parameters that store the current length and maximum length of a formal parameter.

---

---

**Note:** With a parameter of type RAW or LONG RAW, you must use the property LENGTH. Also, if that parameter is IN OUT and NULL or OUT and NULL, then you must set the length of the corresponding C parameter to zero.

---

---

For IN parameters, LENGTH is passed by value (unless you specify BY REFERENCE) and is read-only. For OUT, IN OUT, and RETURN parameters, LENGTH is passed by reference.

As mentioned above, MAXLEN does not apply to IN parameters. For OUT, IN OUT, and RETURN parameters, MAXLEN is passed by reference and is read-only.

## CHARSETID and CHARSETFORM

Oracle provides national language support, which lets you process single-byte and multi-byte character data and convert between character sets. It also lets your applications run in different language environments.

The properties CHARSETID and CHARSETFORM identify the non-default character set from which the character data being passed was formed. With CHAR, CLOB, and VARCHAR2 parameters, you can use CHARSETID and CHARSETFORM to pass the character set ID and form to the external routine.

For IN parameters, CHARSETID and CHARSETFORM are passed by value (unless you specify BY REFERENCE) and are read-only (even if you specify BY REFERENCE). For OUT, IN OUT, and RETURN parameters, CHARSETID and CHARSETFORM are passed by reference and are read-only.

The OCI attribute names for these properties are OCI\_ATTR\_CHARSET\_ID and OCI\_ATTR\_CHARSET\_FORM.

**See Also:** For more information about using NLS data with the OCI, see *Oracle Call Interface Programmer's Guide* and the *Oracle8i National Language Support Guide*.

## Repositioning Parameters

Remember, each formal parameter of the external routine must have a corresponding parameter in the PARAMETERS clause. Their positions can differ, because PL/SQL associates them by name, not by position. However, the PARAMETERS clause and the C prototype for the external routine must have the same number of parameters, and they must be in the same order.

## Using SELF

SELF is the always-present argument of an object type's member function or procedure, namely the object instance itself. In most cases, this argument is implicit and is not listed in the argument list of the PL/SQL procedure. However, SELF must be explicitly specified as an argument of the PARAMETERS clause.

For example, assume that a user wants to create a Person object, consisting of a person's name and date of birth, and then further a table of this object type. The user would eventually like to determine the age of each Person in this table.

---

---

**Note:** You may need to set up data structures similar to the following for certain examples to work:

```
CONNECT system/manager
GRANT CONNECT,RESOURCE,CREATE LIBRARY TO scott IDENTIFIED BY
tiger;
CONNECT scott/tiger
CREATE OR REPLACE LIBRARY agelib UNTRUSTED IS
  '/tmp/scott1.so';
```

This example is only for Solaris; other libraries and include paths might be needed for other platforms.

---

---

In SQL\*Plus, the Person object type can be created by:

```
CREATE OR REPLACE TYPE Person1_typ AS OBJECT
( Name      VARCHAR2(30),
  B_date    DATE,
  MEMBER FUNCTION calcAge_func RETURN NUMBER,
  PRAGMA RESTRICT_REFERENCES(calcAge_func, WNDS)
);
```

Normally, the member function would be implemented in PL/SQL, but for this example, we make it an external procedure. To realize this, the body of the member function is declared as follows:

```
CREATE OR REPLACE TYPE BODY Person1_typ AS
  MEMBER FUNCTION calcAge_func RETURN NUMBER
  AS LANGUAGE C
  NAME "age"
  LIBRARY agelib
  WITH CONTEXT
  PARAMETERS
  ( CONTEXT,
```

```

        SELF,
        SELF INDICATOR STRUCT,
        SELF TDO,
        RETURN INDICATOR
    );
END;
```

Notice that the `calcAge_func` member function doesn't take any arguments, but only returns a number. A member function is always invoked on an instance of the associated object type. The object instance itself always is an implicit argument of the member function. To refer to the implicit argument, the `SELF` keyword is used. This is incorporated into the external procedure syntax by supporting references to `SELF` in the parameters clause.

Now the matching table is created and populated.

```

CREATE TABLE Person_tab OF Person1_typ;

INSERT INTO Person_tab VALUES
    ('SCOTT', TO_DATE('14-MAY-85'));

INSERT INTO Person_tab VALUES
    ('TIGER', TO_DATE('22-DEC-71'));
```

Finally, we retrieve the information of interest from the table.

```
SELECT p.name, p.b_date, p.calcAge_func() FROM Person_tab p;
```

NAME	B_DATE	P.CALCAGE_
SCOTT	14-MAY-85	0
TIGER	22-DEC-71	0

Sample C code, implementing the "external" member function, and the Object-Type-Translator (OTT)-generated struct definitions are included below.

```

#include <oci.h>

struct PERSON
{
    OCIStrng  *NAME;
    OCIDate   B_DATE;
};
typedef struct PERSON PERSON;

struct PERSON_ind
```

```

{
    OCIInd    _atomic;
    OCIInd    NAME;
    OCIInd    B_DATE;
};
typedef struct PERSON_ind PERSON_ind;

OCINumber *age (ctx, person_obj, person_obj_ind, tdo, ret_ind)
OCIExtProcContext *ctx;
PERSON          *person_obj;
PERSON_ind      *person_obj_ind;
OCIType         *tdo;
OCIInd          *ret_ind;
{
    sword      err;
    text       errbuf[512];
    OCIEnv     *envh;
    OCISvcCtx  *svch;
    OCIError   *errh;
    OCINumber  *age;
    int        inum = 0;
    sword      status;

    /* get OCI Environment */
    err = OCIExtProcGetEnv( ctx, &envh, &svch, &errh );

    /* initialize return age to 0 */
    age = (OCINumber *)OCIExtProcAllocCallMemory(ctx, sizeof(OCINumber));
    status = OCINumberFromInt(errh, &inum, sizeof(inum), OCI_NUMBER_SIGNED,
                              age);
    if (status != OCI_SUCCESS)
    {
        OCIExtProcRaiseExcp(ctx, (int)1476);
        return (age);
    }

    /* return NULL if the person object is null or the birthdate is null */
    if ( person_obj_ind->_atomic == OCI_IND_NULL ||
        person_obj_ind->B_DATE == OCI_IND_NULL )
    {
        *ret_ind = OCI_IND_NULL;
        return (age);
    }

    /* The actual implementation to calculate the age is left to the reader,

```

```
        but an easy way of doing this is a callback of the form:
        select trunc(months_between(sysdate, person_obj->b_date) / 12)
        from dual;
    */
    *ret_ind = OCI_IND_NOTNULL;
    return (age);
}
```

### Passing Parameters by Reference

In C, you can pass IN scalar parameters by value (the value of the parameter is passed) or by reference (a pointer to the value is passed). When an external routine expects a pointer to a scalar, specify BY REFERENCE phrase to pass the parameter by reference:

```
CREATE OR REPLACE PROCEDURE findRoot_proc (
    x IN REAL)
AS LANGUAGE C
    LIBRARY c_utils
    NAME "C_findRoot"
    PARAMETERS (
        x BY REFERENCE);
```

In this case, the C prototype would be:

```
void C_findRoot(float *x);
```

This is rather than the default, which would be used when there is no PARAMETERS clause:

```
void C_findRoot(float x);
```

### WITH CONTEXT

By including the WITH CONTEXT clause, you can give an external routine access to information about parameters, exceptions, memory allocation, and the user environment. The WITH CONTEXT clause specifies that a context pointer will be passed to the external routine. For example, if you write the following PL/SQL function:

```
CREATE OR REPLACE FUNCTION getNum_func (
    x IN REAL)
RETURN BINARY_INTEGER AS LANGUAGE C
    LIBRARY c_utils
    NAME "C_getNum"
    WITH CONTEXT
```

```
PARAMETERS (
    CONTEXT,
    x BY REFERENCE,
    RETURN INDICATOR);
```

Then, the C prototype would be:

```
int C_getNum(
    OCIExtProcContext *with_context,
    float *x,
    short *retind);
```

The context data structure is opaque to the external routine; but, is available to service routines called by the external routine.

If you also include the `PARAMETERS` clause, then you must specify the parameter `CONTEXT`, which shows the position of the context pointer in the parameter list. If you omit the `PARAMETERS` clause, then the context pointer is the first parameter passed to the external routine.

### Inter-Language Parameter Mode Mappings

PL/SQL supports the `IN`, `IN OUT`, and `OUT` parameter modes, as well as the `RETURN` clause for routines returning values.

**See Also:** *Oracle8i Java Stored Procedures Developer's Guide*

Rules for PL/SQL and C Parameter Modes are listed above.

## Executing External Routines: the CALL Statement

Now that your Java class method, or external C routine, has been published, you are ready to invoke it.

Do not call an external routine directly. Instead, call the PL/SQL subprogram that published the external routine. Such calls, which you code in the usual way, can appear in the following:

- Anonymous blocks
- Stand-alone and packaged subprograms
- Methods of an object type
- Database triggers

- SQL statements (calls to packaged functions only).

Although the `CALL` statement, described below, is confined to `SELECTs`, it can appear in either the `WHERE` clause or the `SELECT` list.

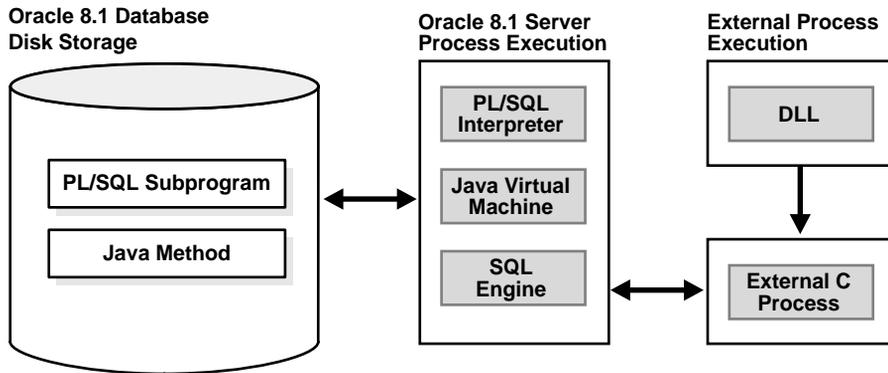
---

**Note:** To call a packaged function from SQL statements, you must use the pragma `RESTRICT_REFERENCES`, which asserts the purity level of the function (the extent to which the function is free of side effects). PL/SQL cannot check the purity level of the corresponding external routine. Therefore, make sure that the routine does not violate the pragma. Otherwise, you might get unexpected results.

---

Any PL/SQL block or subprogram executing on the server side, or on the client side, (for example, in a tool such as Oracle Forms) can call an external procedure. On the server side, the external procedure runs in a separate process address space, which safeguards your database. [Figure 10–1](#) shows how Oracle8 and external routines interact.

**Figure 10–1 Oracle8 and External Routines**



## Preliminaries

Before you call your external routine, you might want to make sure you understand the execution environment. Specifically, you might be interested in privileges, permissions, and synonyms.

## Privileges

When external routines are called via `CALL` specification's, they execute with *definer's privileges*, rather than with the privileges of their invoker.

An invoker's-privileges program is not bound to a particular schema. It executes at the calling site and accesses database items (such as tables and views) with the caller's visibility and permissions. However, a definer's privileges program is bound to the schema in which it is defined. It executes at the defining site, in the definer's schema, and accesses database items with the definer's visibility and permissions.

## Managing Permissions

---

---

**Note:** You may need to set up the following data structures for certain examples to work:

```
CONNECT system/manager
GRANT CREATE ANY DIRECTORY to scott;
CONNECT scott/tiger
CREATE OR REPLACE DIRECTORY bfile_dir AS '/tmp';
CREATE OR REPLACE JAVA RESOURCE NAMED "appImages" USING BFILE
(bfile_dir,'bfile_audio');
```

---

---

To call external routines, a user must have the `EXECUTE` privilege on the call specification and on any resources used by the routine.

In `SQL*Plus`, you can use the `GRANT` and `REVOKE` data control statements to manage permissions. For example:

```
GRANT EXECUTE ON plsToJ_demoExternal_proc TO Public;
REVOKE EXECUTE ON plsToJ_demoExternal_proc FROM Public;
GRANT EXECUTE ON JAVA RESOURCE "appImages" TO Public;
GRANT EXECUTE ON plsToJ_demoExternal_proc TO Scott;
REVOKE EXECUTE ON plsToJ_demoExternal_proc FROM Scott;
```

---

---

**See Also:**

- *Oracle8i SQL Reference*
  - *Oracle8i Java Stored Procedures Developer's Guide*
- 
- 

## Creating Synonyms

For convenience, you or your DBA can create synonyms for external routines using the `CREATE [PUBLIC] SYNONYM` statement. In the example below, your DBA creates a public synonym, which is accessible to all users. If `PUBLIC` is not specified, then the synonym is private and accessible only within its schema.

```
CREATE PUBLIC SYNONYM Rfac FOR Scott.RecursiveFactorial;
```

## CALL Statement Syntax

Invoke the external routine by means of the SQL `CALL` statement. You can execute the `CALL` statement interactively from SQL\*Plus. The syntax is:

```
CALL [schema.][{object_type_name | package_name}]routine_name[@dblink_name]
    [(parameter_list)] [INTO :host_variable][INDICATOR][:indicator_variable];
```

This is essentially the same as executing a routine `foo()` using a SQL statement of the form "`SELECT foo(...) FROM dual,`" except that the overhead associated with performing the `SELECT` is not incurred.

For example, here is an anonymous PL/SQL block which uses dynamic SQL to call `plsToC_demoExternal_proc`, which we published above. PL/SQL passes three parameters to the external C routine `C_demoExternal_proc`.

```
DECLARE
    xx NUMBER(4);
    yy VARCHAR2(10);
    zz DATE;
BEGIN
    EXECUTE IMMEDIATE 'CALL plsToC_demoExternal_proc(:xxx, :yyy, :zzz)' USING
xx,yy,zz;
END;
```

The semantics of the `CALL` statement is identical to the that of an equivalent `BEGIN..END` block.

---



---

**Note:** CALL is the only SQL statement that cannot be put, by itself, in a PL/SQL BEGIN...END block. It can be part of an EXECUTE IMMEDIATE statement within a BEGIN...END block.

---



---

## Calling Java Class Methods

Here is how you would call the `J_calcFactorial` class method published earlier. First, declare and initialize two SQL\*Plus host variables, as follows:

```
VARIABLE x NUMBER
VARIABLE y NUMBER
EXECUTE :x := 5;
```

Now, call `J_calcFactorial`:

```
CALL J_calcFactorial (:x) INTO :y;
PRINT y
```

The result:

```
Y
-----
   120
```

**See Also:** *Oracle8i Java Stored Procedures Developer's Guide*

## Calling External C Routines

To call an external C routine, PL/SQL must know in which DLL it resides. To do this, the PL/SQL engine looks in the data dictionary for the library alias mentioned in the `AS LANGUAGE` clause. Oracle looks for the filename associated with the DLL contained in that library.

Next, PL/SQL alerts a Listener process which, in turn, spawns a session-specific agent named `extproc`. The Listener hands over the connection to `extproc`, and PL/SQL passes to `extproc` the name of the DLL, the name of the external routine, and any parameters.

Then, `extproc` loads the DLL and runs the external routine. Also, `extproc` handles service calls (such as raising an exception) and callbacks to the Oracle server. Finally, `extproc` passes to PL/SQL any values returned by the external routine.

---

---

**Note:** Although some DLL caching takes place, there is no guarantee that your DLL will remain in the cache; therefore, do not store global variables in your DLL.

---

---

After the external routine completes, `extproc` remains active throughout your Oracle session; when you log off, `extproc` is killed. Consequently, you incur the cost of launching `extproc` only once, no matter how many calls you make. Still, you should call an external routine only when the computational benefits outweigh the cost.

---

---

**Note:** The Listener, using the information in the `tnsnames.ora` and `listener.ora` files, must start `extproc` on the machine that runs the Oracle server. Starting `extproc` on a different machine is not supported.

---

---

Here, we call PL/SQL function `plsCallsCdivisor_func`, which we published above, from an anonymous block. PL/SQL passes the two integer parameters to external function `Cdivisor_func`, which returns their greatest common divisor.

```
DECLARE
  g    BINARY_INTEGER;
  a    BINARY_INTEGER;
  b    BINARY_INTEGER;
CALL plsCallsCdivisor_func(a, b);
IF g IN (2,4,8) THEN ...
```

## Errors and Exceptions

### Generic Compile Time Call specification Errors

The PL/SQL compiler raises compile time errors if the following conditions are detected in the syntax:

- An `AS EXTERNAL` call specification is found in a `TYPE` or `PACKAGE` specification.

## Java Exception Handling

**See Also:** *Oracle8i Java Stored Procedures Developer's Guide*

## C Exception Handling

C programs can raise exceptions through the `OCIExtProc...` functions.

## Using Service Routines with External C Routines

When called from an external routine, a service routine can raise exceptions, allocate memory, and invoke OCI handles for callbacks to the server. To use a **service routine**, you must specify the `WITH CONTEXT` clause, which lets you pass a context structure to the external routine. The context structure is declared in header file `ociextp.h` as follows:

```
typedef struct OCIExtProcContext OCIExtProcContext;
```

---

---

**Note:** `ociextp.h` is located in `$ORACLE_HOME/plsql/public` on UNIX.

---

---

### **OCIExtProcAllocCallMemory**

This service routine allocates *n* bytes of memory for the duration of the external routine call. Any memory allocated by the function is freed automatically as soon as control returns to PL/SQL.

---

---

**Note:** The external routine does not need to (and should not) call the C function `free()` to free memory allocated by this service routine as this is handled automatically.

---

---

The C prototype for this function is as follows:

```
dvoid *OCIExtProcAllocCallMemory(  
    OCIExtProcContext *with_context,  
    size_t amount);
```

The parameters `with_context` and `amount` are the context pointer and number of bytes to allocate, respectively. The function returns an untyped pointer to the allocated memory. A return value of zero indicates failure.

In SQL\*Plus, suppose you publish external function `plsToC_concat_func`, as follows:

---



---

**Note:** You may need to set up data structures similar to the following for certain examples to work:

```
CONNECT system/manager
DROP USER y CASCADE;
GRANT CONNECT,RESOURCE,CREATE LIBRARY TO y IDENTIFIED BY y;
CONNECT y/y
CREATE LIBRARY stringlib AS
'/private/varora/ilmswork/Cexamples/john2.so';
```

---



---

```
CREATE OR REPLACE FUNCTION plsToC_concat_func (
    str1 IN VARCHAR2,
    str2 IN VARCHAR2)
RETURN VARCHAR2 AS LANGUAGE C
NAME "concat"
LIBRARY stringlib
WITH CONTEXT
PARAMETERS (
CONTEXT,
str1  STRING,
str1  INDICATOR short,
str2  STRING,
str2  INDICATOR short,
RETURN INDICATOR short,
RETURN LENGTH short,
RETURN STRING);
```

When called, `C_concat` concatenates two strings, then returns the result:

```
select plsToC_concat_func('hello ', 'world') from dual;
```

```
PLSTOC_CONCAT_FUNC('HELLO ','WORLD')
```

```
-----
hello world
```

If either string is `NULL`, then the result is also `NULL`. As the following example shows, `C_concat` uses `OCIExtProcAllocCallMemory` to allocate memory for the result string:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```

#include <oci.h>
#include <ociextp.h>

char *concat(ctx, str1, str1_i, str2, str2_i, ret_i, ret_l)
OCIExtProcContext *ctx;
char *str1;
short str1_i;
char *str2;
short str2_i;
short *ret_i;
short *ret_l;
{
    char *tmp;
    short len;
    /* Check for null inputs. */
    if ((str1_i == OCI_IND_NULL) || (str2_i == OCI_IND_NULL))
    {
        *ret_i = (short)OCI_IND_NULL;
        /* PL/SQL has no notion of a NULL ptr, so return a zero-byte string. */
        tmp = OCIExtProcAllocCallMemory(ctx, 1);
        tmp[0] = '\0';
        return(tmp);
    }
    /* Allocate memory for result string, including NULL terminator. */
    len = strlen(str1) + strlen(str2);
    tmp = OCIExtProcAllocCallMemory(ctx, len + 1);

    strcpy(tmp, str1);
    strcat(tmp, str2);

    /* Set NULL indicator and length. */
    *ret_i = (short)OCI_IND_NOTNULL;
    *ret_l = len;
    /* Return pointer, which PL/SQL frees later. */
    return(tmp);
}

#ifdef LATER
static void checkerr (/*_ OCIError *errhp, sword status _*/);

void checkerr(errhp, status)
OCIError *errhp;
sword status;
{
    text errbuf[512];

```

```
sb4 errcode = 0;

switch (status)
{
case OCI_SUCCESS:
    break;
case OCI_SUCCESS_WITH_INFO:
    (void) printf("Error - OCI_SUCCESS_WITH_INFO\n");
    break;
case OCI_NEED_DATA:
    (void) printf("Error - OCI_NEED_DATA\n");
    break;
case OCI_NO_DATA:
    (void) printf("Error - OCI_NODATA\n");
    break;
case OCI_ERROR:
    (void) OCIErrorGet((dvoid *)errhp, (ub4) 1, (text *) NULL, &errcode,
                      errbuf, (ub4) sizeof(errbuf), OCI_HTYPE_ERROR);
    (void) printf("Error - %.*s\n", 512, errbuf);
    break;
case OCI_INVALID_HANDLE:
    (void) printf("Error - OCI_INVALID_HANDLE\n");
    break;
case OCI_STILL_EXECUTING:
    (void) printf("Error - OCI_STILL_EXECUTE\n");
    break;
case OCI_CONTINUE:
    (void) printf("Error - OCI_CONTINUE\n");
    break;
default:
    break;
}
}

char *concat(ctx, str1, str1_i, str2, str2_i, ret_i, ret_l)
OCIExtProcContext *ctx;
char *str1;
short str1_i;
char *str2;
short str2_i;
short *ret_i;
short *ret_l;
{
    char *tmp;
    short len;
```

```

/* Check for null inputs. */
if ((str1_i == OCI_IND_NULL) || (str2_i == OCI_IND_NULL))
{
    *ret_i = (short)OCI_IND_NULL;
    /* PL/SQL has no notion of a NULL ptr, so return a zero-byte string. */
    tmp = OCIEExtProcAllocCallMemory(ctx, 1);
    tmp[0] = '\0';
    return(tmp);
}
/* Allocate memory for result string, including NULL terminator. */
len = strlen(str1) + strlen(str2);
tmp = OCIEExtProcAllocCallMemory(ctx, len + 1);

strcpy(tmp, str1);
strcat(tmp, str2);

/* Set NULL indicator and length. */
*ret_i = (short)OCI_IND_NOTNULL;
*ret_l = len;
/* Return pointer, which PL/SQL frees later. */
return(tmp);
}

/*=====*/
int main(char *argv, int argc)
{
    OCIEExtProcContext *ctx;
    char *str1;
    short str1_i;
    char *str2;
    short str2_i;
    short *ret_i;
    short *ret_l;
    /* OCI Handles */
    OCIEEnv *envhp;
    OCIServer *srvhp;
    OCISvcCtx *svchp;
    OCIError *errhp;
    OCISession *authp;
    OCISstmt *stmthp;
    OCILobLocator *clob, *blob;
    OCILobLocator *lob_loc;

    /* Initialize and Logon */
    (void) OCIInitialize((ub4) OCI_DEFAULT, (dvoid *)0,

```

```
(dvoid * (*)(dvoid *, size_t)) 0,  
(dvoid * (*)(dvoid *, dvoid *, size_t))0,  
(void (*)(dvoid *, dvoid *)) 0 );  
  
(void) OCIEnvInit( (OCIEnv **) &envhp,  
    OCI_DEFAULT, (size_t) 0,  
    (dvoid **) 0 );  
  
(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, OCI_HTYPE_ERROR,  
    (size_t) 0, (dvoid **) 0);  
  
/* Server contexts */  
(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, OCI_HTYPE_SERVER,  
    (size_t) 0, (dvoid **) 0);  
  
/* Service context */  
(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, OCI_HTYPE_SVCCTX,  
    (size_t) 0, (dvoid **) 0);  
  
/* Attach to Oracle */  
(void) OCIServerAttach( srvhp, errhp, (text *)"", strlen(""), 0);  
  
/* Set attribute server context in the service context */  
(void) OCIAttrSet ((dvoid *) svchp, OCI_HTYPE_SVCCTX,  
    (dvoid *)srvhp, (ub4) 0,  
    OCI_ATTR_SERVER, (OCIError *) errhp);  
  
(void) OCIHandleAlloc((dvoid *) envhp,  
    (dvoid **)&authp, (ub4) OCI_HTYPE_SESSION,  
    (size_t) 0, (dvoid **) 0);  
  
(void) OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,  
    (dvoid *) "samp", (ub4)4,  
    (ub4) OCI_ATTR_USERNAME, errhp);  
  
(void) OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,  
    (dvoid *) "samp", (ub4) 4,  
    (ub4) OCI_ATTR_PASSWORD, errhp);  
  
/* Begin a User Session */  
checkerr(errhp, OCISessionBegin ( svchp, errhp, authp, OCI_CRED_RDBMS,  
    (ub4) OCI_DEFAULT));  
  
(void) OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,  
    (dvoid *) authp, (ub4) 0,
```

```

        (ub4) OCI_ATTR_SESSION, errhp);

/* -----User Logged In-----*/
printf ("user logged in \n");

/* allocate a statement handle */
checkerr(errhp, OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
                               OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));

checkerr(errhp, OCIDescriptorAlloc((dvoid *)envhp, (dvoid **) &Lob_loc,
                                   (ub4) OCI_DTYPE_LOB,
                                   (size_t) 0, (dvoid **) 0));

/* ----- subroutine called here-----*/
printf ("calling concat...\n");
concat(ctx, str1, str1_i, str2, str2_i, ret_i, ret_l);

return 0;
}

#endif

```

### OCIExtProcRaiseExcp

This service routine raises a predefined exception, which must have a valid Oracle error number in the range 1...32767. After doing any necessary cleanup, your external must return immediately. (No values are assigned to OUT or IN OUT parameters.) The C prototype for this function follows:

```

int OCIExtProcRaiseExcp(
    OCIExtProcContext *with_context,
    size_t errnum);

```

The parameters `with_context` and `error_number` are the context pointer and Oracle error number. The return values `OCIEXTPROC_SUCCESS` and `OCIEXTPROC_ERROR` indicate success or failure.

In SQL\*Plus, suppose you publish external routine `plsTo_divide_proc`, as follows:

```

CREATE OR REPLACE PROCEDURE plsTo_divide_proc (
    dividend IN BINARY_INTEGER,
    divisor  IN BINARY_INTEGER,
    result   OUT FLOAT)
AS LANGUAGE C
    NAME "C_divide"

```

```
LIBRARY MathLib
WITH CONTEXT
PARAMETERS (
    CONTEXT,
    dividend INT,
    divisor INT,
    result FLOAT);
```

When called, `C_divide` finds the quotient of two numbers. As the following example shows, if the divisor is zero, `C_divide` uses `OCIExtProcRaiseExcp` to raise the predefined exception `ZERO_DIVIDE`:

```
void C_divide (ctx, dividend, divisor, result)
OCIExtProcContext *ctx;
int dividend;
int divisor;
float *result;
{
    /* Check for zero divisor. */
    if (divisor == (int)0)
    {
        /* Raise exception ZERO_DIVIDE, which is Oracle error 1476. */
        if (OCIExtProcRaiseExcp(ctx, (int)1476) == OCIEXTPROC_SUCCESS)
        {
            return;
        }
        else
        {
            /* Incorrect parameters were passed. */
            assert(0);
        }
    }
    *result = (float)dividend / (float)divisor;
}
```

### **OCIExtProcRaiseExcpWithMsg**

This service routine raises a user-defined exception and returns a user-defined error message. The C prototype for this function follows:

```
int OCIExtProcRaiseExcpWithMsg(
    OCIExtProcContext *with_context,
    size_t error_number,
    text *error_message,
    size_t len);
```

The parameters `with_context`, `error_number`, and `error_message` are the context pointer, Oracle error number, and error message text. The parameter `len` stores the length of the error message. If the message is a null-terminated string, then `len` is zero. The return values `OCIEXTPROC_SUCCESS` and `OCIEXTPROC_ERROR` indicate success or failure.

In the previous example, we published external routine `plsTo_divide_proc`. In the example below, you use a different implementation. With this version, if the divisor is zero, then `C_divide` uses `OCIExtProcRaiseExcpWithMsg` to raise a user-defined exception:

```
void C_divide (ctx, dividend, divisor, result)
OCIExtProcContext *ctx;
int  dividend;
int  divisor;
float *result;
    /* Check for zero divisor. */
    if (divisor == (int)0)
    {
        /* Raise a user-defined exception, which is Oracle error 20100,
           and return a null-terminated error message. */
        if (OCIExtProcRaiseExcpWithMsg(ctx, (int)20100,
            "divisor is zero", 0) == OCIEXTPROC_SUCCESS)
        {
            return;
        }
        else
        {
            /* Incorrect parameters were passed. */
            assert(0);
        }
    }
    *result = dividend / divisor;
}
```

## Doing Callbacks with External C Routines

### OCIExtProcGetEnv

This service routine enables OCI callbacks to the database during an external routine call. It is only used for callbacks, and, furthermore, it is the only callback routine used. If you use the OCI handles obtained by this function for standard OCI

calls, then the handles establish a new connection to the database and cannot be used for callbacks in the same transaction. In other words, during an external routine call, you can use OCI handles for callbacks or a new connection but not for both.

The C prototype for this function follows:

```
sword OCIExtProcGetEnv ( OCIExtProcContext *with_context,
                        OCIEnv envh,
                        OCISvcCtx svch,
                        OCIError errh )
```

The parameter `with_context` is the context pointer, and the parameters `envh`, `svch`, and `errh` are the OCI environment, service, and error handles, respectively. The return values `OCIEXTPROC_SUCCESS` and `OCIEXTPROC_ERROR` indicate success or failure.

Both External C routines and Java class methods can call-back to the database to do SQL operations. For a working example, see "[Demo Program](#)" on page 10-48.

Java exceptions:

**See Also:** *Oracle8i Java Stored Procedures Developer's Guide*

---

---

**Note:** Callbacks are not necessarily a same-session phenomenon; you may execute an SQL statement in a different session via `OCIlogon`.

---

---

An external C routine executing on the Oracle server can call a service routine to obtain OCI environment and service handles. With the OCI, you can use callbacks to execute SQL statements and PL/SQL subprograms, fetch data, and manipulate LOBs. Callbacks and external routines operate in the same user session and transaction context, and so have the same user privileges.

In SQL\*Plus, suppose you run the following script:

```
CREATE TABLE Emp_tab (empno NUMBER(10))

CREATE PROCEDURE plsToC_insertIntoEmpTab_proc (
    empno BINARY_INTEGER)
AS LANGUAGE C
    NAME "C_insertEmpTab"
    LIBRARY insert_lib
```

```

WITH CONTEXT
PARAMETERS (
    CONTEXT,
    empno LONG);

```

Later, you might call service routine `OCIExtProcGetEnv` from external routine `plsToC_insertIntoEmpTab_proc`, as follows:

```

#include <stdio.h>
#include <stdlib.h>
#include <oratypes.h>
#include <oci.h> /* includes ociextp.h */
...
void C_insertIntoEmpTab (ctx, empno)
OCIExtProcContext *ctx;
long empno;
{
    OCIEnv *envhp;
    OCISvcCtx *svchp;
    OCIError *errhp;
    int err;
    ...
    err = OCIExtProcGetEnv(ctx, &envhp, &svchp, &errhp);
    ...
}

```

If you do not use callbacks, you do not need to include `oci.h`; instead, just include `ociextp.h`.

## Object Support for OCI Callbacks

To execute object-related callbacks from your external routines, the OCI environment in the `extproc` agent is now fully initialized in object mode. You retrieve handles to this environment with the `OCIExtProcGetEnv()` routine.

The object runtime environment lets you use static, as well as dynamic, object support provided by OCI. To utilize static support, use the OTT to generate C structs for the appropriate object types, and then use conventional C code to access the objects' attributes.

For those objects whose types are unknown at external procedure creation time, an alternative, dynamic, way of accessing objects is first to invoke `OCIDescribeAny()` to obtain attribute and method information about the type. Then,

`OCIObjectGetAttr()` and `OCIObjectSetAttr()` can be called to retrieve and set attribute values.

Because the current external routine model is stateless, `OCIExtProcGetEnv()` must be called in every external routine that wants to execute callbacks, or invoke `OCIExtProc...()` service routines. After every external routine invocation, the callback mechanism is cleaned up and all OCI handles are freed.

## Restrictions on Callbacks

With callbacks, the following SQL commands and OCI routines are not supported:

- Transaction control commands such as `COMMIT`
- Data definition commands such as `CREATE`
- The following object-oriented OCI routines:

- `OCIObjectNew`
- `OCIObjectPin`
- `OCIObjectUnpin`
- `OCIObjectPinCountReset`
- `OCIObjectLock`
- `OCIObjectMarkUpdate`
- `OCIObjectUnmark`
- `OCIObjectUnmarkByRef`
- `OCIObjectAlwaysLatest`
- `OCIObjectNotAlwaysLatest`
- `OCIObjectMarkDeleteByRef`
- `OCIObjectMarkDelete`
- `OCIObjectFlush`
- `OCIObjectFlushRefresh`
- `OCIObjectGetTypeRef`
- `OCIObjectGetObjectRef`
- `OCIObjectExists`
- `OCIObjectIsLocked`
- `OCIObjectIsDirtied`
- `OCIObjectIsLoaded`
- `OCIObjectRefresh`
- `OCIObjectPinTable`
- `OCIObjectArrayPin`
- `OCICacheFlush,`
- `OCICacheFlushRefresh,`
- `OCICacheRefresh`
- `OCICacheUnpin`
- `OCICacheFree`

```

OCICacheUnmark
OCICacheGetObjects
OCICacheRegister

```

- Polling-mode OCI routines such as `OCIGetPieceInfo`
- The following OCI routines:

```

OCIEnvInit
OCIInitialize
OCIPasswordChange
OCIserverAttach
OCIserverDetach
OCISessionBegin
OCISessionEnd
OCISvcCtxToLda
OCITransCommit
OCITransDetach
OCITransRollback
OCITransStart

```

Also, with OCI routine `OCIHandleAlloc`, the following handle types are not supported:

```

OCI_HTYPE_SERVER
OCI_HTYPE_SESSION
OCI_HTYPE_SVCCTX
OCI_HTYPE_TRANS

```

## Debugging External Routines

**See Also:** *Oracle8i Java Stored Procedures Developer's Guide*

Usually, when an external routine fails, its prototype is faulty. In other words, the prototype does not match the one generated internally by PL/SQL. This can happen if you specify an incompatible C datatype. For example, to pass an OUT parameter of type REAL, you must specify `float *`. Specifying `float`, `double *`, or any other C datatype will result in a mismatch.

In such cases, you might get:

```
lost RPC connection to external routine agent
```

This error, which means that agent `extproc` terminated abnormally because the external routine caused a core dump. To avoid errors when declaring C prototype parameters, refer to the tables above.

### Using Package `DEBUG_EXTPROC`

To help you debug external routines, PL/SQL provides the utility package `DEBUG_EXTPROC`. To install the package, run the script `dbgextp.sql` which you can find in the PL/SQL demo directory. (For the location of the directory, see your Oracle Installation or User's Guide.)

To use the package, follow the instructions in `dbgextp.sql`. Your Oracle account must have `EXECUTE` privileges on the package and `CREATE LIBRARY` privileges.

---

---

**Note:** `DEBUG_EXTPROC` works only on platforms with debuggers that can attach to a running process.

---

---

## Demo Program

Also in the PL/SQL demo directory is the script `extproc.sql`, which demonstrates the calling of an external routine. The companion file `extproc.c` contains the C source code for the external routine.

To run the demo, follow the instructions in `extproc.sql`. You must use the `SCOTT/TIGER` account, which must have `CREATE LIBRARY` privileges.

## Guidelines for External C Routines

### Handling Global And Static Variables

**Global Variables** A global variable is declared outside of a function, and its value is shared by all functions of a program. In case of external routines, this means that all functions in a DLL share the value of the global. The usage of global variables is discouraged for two reasons:

- **Threading:** In the current non-threaded configuration of the `extproc` process, there is only one function active at a time. In the future, however, Oracle might thread the `extproc` process, which would mean that multiple functions can be active at the same time. In that case, it is possible that two or more functions concurrently would try to access the global variable with unsuccessful results.

- **DLL caching:** Global variables are also used to store data that is intended to persist beyond the lifetime of a function. For example, consider two functions *func1()* and *func2()* trying to pass data to each other. Because of the DLL caching feature, it is possible that after *func1()*'s completion, the DLL will be unloaded, which results in all global variables losing their values. When *func2()* is executed, the DLL is reloaded, and all globals are initialized to 0, which will be inconsistent with their values at the completion of *func1()*.

**Static Variables** There are two types of static variables: external and internal. An external static variable is a special case of a global variable, so its usage is discouraged for the above two reasons. Internal static variables are local to a particular function, but remain in existence rather than coming and going each time the function is activated. Therefore, they provide private, permanent storage within a single function. These variables are used to pass on data to subsequent invocation of the same function. But, because of the DLL caching feature mentioned above, the DLL might be unloaded and reloaded between invocations, which means that the internal static variable would lose its value.

**See Also:** For help in creating a dynamic link library, look in the RDBMS subdirectory */public*, where a template *makefile* can be found.

## Call specification and CALLing Guidelines

When calling external routines:

- Never write to `IN` parameters or overflow the capacity of `OUT` parameters. (PL/SQL does no run time checks for these error conditions.)
- Never read an `OUT` parameter or a function result.
- Always assign a value to `IN` `OUT` and `OUT` parameters and to function results. Otherwise, your external routine will not return successfully.
- If you include the `WITH CONTEXT` and `PARAMETERS` clauses, then you must specify the parameter `CONTEXT`, which shows the position of the context pointer in the parameter list.
- If you include the `PARAMETERS` clause, and if the external routine is a function, then you must specify the parameter `RETURN` in the last position.
- For every formal parameter, there must be a corresponding parameter in the `PARAMETERS` clause. Also, make sure that the datatypes of parameters in the `PARAMETERS` clause are compatible with those in the C prototype, because no implicit conversions are done.

- With a parameter of type RAW or LONG RAW, you must use the property LENGTH. Also, if that parameter is IN OUT or OUT and null, then you must set the length of the corresponding C parameter to zero.

## Restrictions on External C Routines

Currently, the following restrictions apply to external routines:

- This feature is available only on platforms that support DLLs.
- Only C routines and routines callable from C code are supported.
- You cannot pass PL/SQL cursor variables, records, collections, or instances of an object type to an external routine.
- In the LIBRARY subclause, you cannot use a database link to specify a remote library.
- The Listener must start agent `extproc` on the machine that runs the Oracle server. Starting `extproc` on a different machine is not supported.
- The maximum number of parameters that you can pass to a external routine is 128. However, if you pass float or double parameters by value, then the maximum is less than 128. How much less depends on the number of such parameters and your operating system. To get a rough estimate, count each float or double passed by value as two parameters.

---

# Establishing Security Policies

This chapter provides guidance on developing security policies. It includes:

- [Introduction to Security Policies](#)
- [Application Security](#)
- [Fine-Grained Access Control](#)
- [Application Context](#)
- [Authentication Through a Middle Tier](#)
- [Data Encryption](#)

## Introduction to Security Policies

This section provides an introduction to discretionary security policies. It covers:

- [Security Threats and Countermeasures](#)
- [What Discretionary Security Policies Can Cover](#)
- [Features to Use in Establishing Security Policies](#)

## Security Threats and Countermeasures

Organizations should create a written security policy to enumerate the security threats they are trying to guard against, and the specific measures the organization must take. Security threats can be addressed with types of measures:

- Procedural, such as requiring data center employees to display security badges
- Personnel, such as performing background checks or "vetting" key personnel
- Physical, such as securing computers in restricted-access facilities
- Technical, such as implementing strong authentication requirements for critical business systems

Consider whether the appropriate response to a threat is procedural, physical, technical, or personnel-related, or whether the threat can be met by a combination of the above measures.

For example, a possible security threat is disruption of critical business systems caused by a malicious person damaging a computer. A physical response to this threat is to secure key business computers in a locked facility. A procedural response is to create system backups at regular intervals. Personnel measures could include background checks on employees who access or manage key business systems.

Oracle*8i* offers many mechanisms which can implement the *technical* measures of a good security policy.

## What Discretionary Security Policies Can Cover

In addition to requirements unique to your environment, you should design and implement discretionary security policies to determine the following technical issues:

- The level of security at the application level
- System and object privileges
- Database roles
- How to grant and revoke privileges and roles
- How to create, alter, and drop roles
- How to control role use
- Level of granularity of access control
- User attributes which govern access to the database
- Whether to use encryption
- How to implement security in three-tier applications

## Features to Use in Establishing Security Policies

This chapter discusses several elements of Oracle8i which you can use in establishing security policies:

Application Security	Use this feature to attach privileges and roles to each application, while making sure that users do not misuse those roles and privileges when they are not actually using the application.
Application Context	Use this feature to set up session-based attributes securely. For example, you can securely store such user attributes as a user name, employee number, the set of books she is authorized to access, and her position in the management hierarchy. You can then retrieve that information later in the session.
Fine-Grained Access Control	Use this feature to implement security policies at a low level of granularity; for example, to enforce row-level security. Do this by creating security policy functions attached to the table or view on which you have based your application. Then, when a user enters a DML statement on that object, Oracle modifies that statement dynamically and transparently to the user.
Authentication and Auditing Through a Middle Tier	Use this feature to allow each user to be authenticated by a database password, without the overhead of a separate database connection. You can also use this feature to identify the real user who is accessing the database via a middle tier.
Data Encryption	Use this feature to encrypt information, as an extra measure of security.

## Application Security

Draft security policies for each database application. For example, each database application should have one or more application roles that provide different levels of security when executing the application. The application roles can be granted to user roles or directly to specific usernames.

Applications that potentially allow unrestricted SQL statement execution (through tools such as SQL\*Plus) also need security policies that prevent malicious access to confidential or important schema objects.

This section describes the following aspects of application security:

- [Considerations for Using Application-Based Security](#)
- [Tasks of Application Administrators](#)
- [Overview of Roles and Application Privilege Management](#)
- [Associating Privileges with the User's Current Application Role](#)
- [Restricting Application Roles from Tool Users](#)
- [Protecting Database Objects Through Use of Schemas](#)
- [Managing Object Privileges](#)
- [Creating a Role and Protecting Its Use](#)
- [Enabling and Disabling Roles](#)
- [Granting and Revoking System Privileges and Roles](#)
- [Granting and Revoking Schema Object Privileges and Roles](#)
- [Granting to, and Revoking from, the User Group PUBLIC](#)

## Considerations for Using Application-Based Security

There are many issues to consider when you formulate and implement application security. Two of the main considerations are these:

- [Are Application Users Also Database Users?](#)
- [Does Security Enforcement Occur in the Application, or in the Database?](#)

### Are Application Users Also Database Users?

Oracle recommends that, where possible, you build applications in which application users are database users. In this way you can leverage the intrinsic security mechanisms of the database.

For many commercial packaged applications, application users are not database users. For these applications, multiple users authenticate themselves to the application, and the application then connects to the database as a single, highly-privileged user. We will call this the "One Big Application User" model.

Applications built in this fashion generally cannot use many of the intrinsic security features of the database, because the identity of the user is not known to the database.

Examples of features whose use is compromised by the One Big Application User model include:

- **Auditing.** A basic principle of security is accountability through auditing. However, if all actions in the database are done by One Big Application User, then database auditing cannot hold individual users accountable for their actions. The application must implement its own auditing mechanisms to capture individual users' actions.
- **Oracle Advanced Security enhanced authentication.** Strong forms of authentication supported by Oracle Advanced Security (such as Kerberos, DCE, client authentication over SSL, tokens, and so on) cannot be used if the client authenticating to the database is the application, rather than an individual user.
- **Roles.** Roles are assigned to database users. If application users are not database users, then the usefulness of roles is diminished. Applications must then craft their own mechanisms to distinguish between the privileges various application users need to access data within the application.

### **Does Security Enforcement Occur in the Application, or in the Database?**

Oracle recommends that, to the extent possible, applications utilize the security enforcement mechanisms of the database. When security is enforced in the database itself rather than the application, it cannot be bypassed by using another application to access data.

Applications whose users are also database users can either build security into the application, or rely upon intrinsic database security mechanisms such as granular privileges, virtual private database (fine-grained access control with application context), roles, stored procedures, and auditing.

Applications that use the One Big Application User model must build security enforcement into the application rather than using database security mechanisms. In this case, since it is the application—and not the database—which recognizes users, the application must enforce any per-user security measures itself.

If applications enforce their own security mechanisms, it means that each application that accesses data must reimplement security. For example, if an organization implements a new report-writing tool, then the organization must implement security to ensure that users do not get more data access through the report-writing tool than they have in the application itself. Security becomes expensive because organizations must implement the same security policies in multiple applications; each new application requires an expensive reimplementation.

The main shortcoming of application-based security is that security is bypassed if the user bypasses the application to access data. For example, a user who has SQL\*Plus access to the database can execute queries without going through the Human Resources application. The user thus bypasses all the security measures in the application.

## Tasks of Application Administrators

In large database systems with many applications, you may decide to have application administrators. An application administrator is responsible for the following:

- Creating roles for the database application and managing the privileges of each application role
- Creating and managing the objects used by the application
- Maintaining and updating the application code, and Oracle procedures and packages, as necessary

## Overview of Roles and Application Privilege Management

Because most database applications involve many different privileges on many different schema objects, keeping track of which privileges are required for each application can be complex. In addition, authorizing users to run an application can involve many `GRANT` operations. To simplify application privilege management, create a role for each application and grant that role all the privileges a user needs to run the application. In fact, an application might have a number of roles, each granted a specific subset of privileges that allow fewer or more capabilities while running the application.

For example, suppose that every administrative assistant uses the Vacation application to record vacation taken by members of the department. You should:

1. Create a `VACATION` role.
2. Grant all privileges required by the Vacation application to the `VACATION` role.
3. Grant the `VACATION` role to all administrative assistants or to a role named `ADMIN_ASSISTS` (if previously defined).

Grouping application privileges in a role aids privilege management. Consider the following administrative options:

- You can grant the role, rather than many individual privileges, to those users who run the application. Then, as employees change jobs, you need to grant or revoke only one role, rather than many privileges.
- You can change the privileges associated with an application by modifying only the privileges granted to the role, rather than the privileges held by all users of the application.

- You can determine which privileges are necessary to run a particular application by querying the `ROLE_TAB_PRIVS` and `ROLE_SYS_PRIVS` data dictionary views.
- You can determine which users have privileges on which applications by querying the `DBA_ROLE_PRIVS` data dictionary view.

## Associating Privileges with the User's Current Application Role

A single user can use many applications and associated roles. However, you should allow a user to have only the privileges associated with the currently running application role. For example, consider the following scenario:

- The `ORDER` role (for the `ORDER` application) contains the `UPDATE` privilege for the `INVENTORY` table.
- The `INVENTORY` role (for the `INVENTORY` application) contains the `SELECT` privilege for the `INVENTORY` table.
- Several order entry clerks have been granted both the `ORDER` and `INVENTORY` roles.

In this scenario, an order entry clerk who has been granted both roles, can presumably use the privileges of the `ORDER` role when running the `INVENTORY` application to update the `INVENTORY` table. The problem is that updating the `INVENTORY` table is not an authorized action when using the `INVENTORY` application, but only when using the `ORDER` application.

To avoid such problems, consider using either the `SET ROLE` statement or the `SET_ROLE` procedure as explained below.

### SET ROLE Statement

Use a `SET ROLE` statement at the beginning of each application to automatically enable its associated role and, consequently, disable all others. In this way, each application dynamically enables particular privileges for a user only when required.

The `SET ROLE` statement facilitates privilege management because, in addition to letting you control what information a user can access, it allows you to control when a user can access it. In addition, the `SET ROLE` statement keeps users operating in a well-defined privilege domain. If a user gets all privileges from roles, then the user cannot combine these privileges to perform unauthorized operations.

**See Also:** ["Enabling and Disabling Roles"](#) on page 11-24

## SET\_ROLE Procedure

The PL/SQL package `DEMS_SESSION.SET_ROLE` is functionally equivalent to the `SET ROLE` statement in SQL.

A limitation of roles is the inability to `SET ROLE` within a definer's rights procedure. The reason is that, for a definer's rights procedure, the database checks privileges at compilation time, not at execution time. That is, the database verifies that the owner of the procedure has necessary privileges—granted to him directly, not through a role—at the time the procedure is compiled. A `SET ROLE` statement does not work because the role is not enabled at compilation time, when the database checks privileges. At execution time, when the role is to be enabled, the database does not check the owner's privileges; instead, the database merely ensures that a user of the procedure has `EXECUTE` privilege on the procedure.

In cases where the database checks privileges at execution time instead of compilation time, it is possible to issue a `SET ROLE`. Thus, the `DEMS_SESSION.SET_ROLE` command can be called from the following

- Anonymous PL/SQL blocks
- Invoker's rights stored procedures (except those invoked from within definer's rights procedures)

In both the above cases, the database checks privileges at execution time, not at compilation time. Therefore, the database can validate that a user has appropriate privileges (that is, that the user has been granted the role that is being set).

**Note:** If you use `DEMS_SESSION.SET_ROLE` within an invoker's rights procedure, the role remains in effect until you specifically disable it. In keeping with the principle of "least privilege," (that users should have the fewest privileges they need to do their jobs), you should explicitly disable roles set within an invoker's rights procedure, at the end of the procedure.

Because PL/SQL does the security check on SQL when an anonymous block is compiled, `SET_ROLE` will not affect the security role (in other words, it will not affect the roles enabled) for embedded SQL statements or procedure calls.

## Example

---



---

**Note:** You may need to set up data structures for the following example, and certain others, to work. Set up the following:

```
CONNECT system/manager
DROP USER joe CASCADE;
CREATE USER joe IDENTIFIED BY joe;
GRANT CREATE SESSION, RESOURCE, UNLIMITED TABLESPACE TO joe;
GRANT CREATE SESSION, RESOURCE, UNLIMITED TABLESPACE TO scott;
DROP ROLE acct;
CREATE ROLE acct;
GRANT acct TO scott;

CONNECT joe/joe;
CREATE TABLE finance (empno NUMBER);
GRANT SELECT ON finance TO acct;
CONNECT scott/tiger
```

---



---

Suppose you have a role named ACCT that has been granted privileges allowing you to select from table FINANCE in the JOE schema. In this case, the following block fails:

```
DECLARE
    n NUMBER;
BEGIN
    SYS.DBMS_SESSION.SET_ROLE('acct');
    SELECT empno INTO n FROM JOE.FINANCE;
END;
```

The block fails because the security check which verifies that you have the SELECT privilege on table JOE.FINANCE occurs at compile time. At compile time, however, the ACCT role is not yet enabled. The role is not enabled until the block is executed.

The DBMS\_SQL package, by contrast, is not subject to this restriction. When you use this package, the security checks are performed at runtime. Thus, a call to SET\_ROLE would affect the SQL executed using calls to the DBMS\_SQL package. The following block is, therefore, successful:

```
CREATE OR REPLACE PROCEDURE dynSQL_proc IS
    n NUMBER;
BEGIN
    SYS.DBMS_SESSION.SET_ROLE('acct');
    EXECUTE IMMEDIATE 'select empno from joe.finance' INTO n;
```

```
--other calls to SYS.DBMS_SQL  
END;
```

**See Also:** ["Native Dynamic SQL vs. the DBMS\\_SQL Package"](#) on page 8-12.

## Restricting Application Roles from Tool Users

Prebuilt database applications explicitly control the potential actions of a user, including the enabling and disabling of the user's roles while using the application. By contrast, ad hoc query tools, such as SQL\*Plus, allow a user to submit any SQL statement (which may or may not succeed), including the enabling and disabling of any granted role. This can pose a serious security problem.

This section describes:

- [A Potential Problem](#)
- [Limiting Roles Through PRODUCT\\_USER\\_PROFILE](#)
- [Virtual Private Database for Greatest Security](#)
- [Recommended Application Design Practices to Reduce Risk](#)
- [Enable and Disable Roles Promptly](#)
- [Encapsulate Privileges in Stored Procedures](#)
- [Use Role Passwords Unknown to the User](#)
- [Use Application Context and Fine-Grained Access Control](#)

## A Potential Problem

A user of an application could exercise the privileges attached to that application to issue destructive SQL statements against database tables by using an ad hoc tool.

For example, consider the following scenario:

- The Vacation application has a corresponding VACATION role.
- The VACATION role includes the privileges to issue SELECT, INSERT, UPDATE, and DELETE statements against the EMP\_TAB table.
- The Vacation application controls the use of the privileges obtained via the VACATION role (the application controls when statements are issued).

Now, consider a user who has been granted the VACATION role. Suppose, instead of using the Vacation application, the user executes SQL\*Plus. At this point, the user is restricted only by the privileges granted to him explicitly or via roles, including the VACATION role. Because SQL\*Plus is an ad hoc query tool, the user is not restricted to a set of predefined actions, as with designed database applications. The user can query or modify data in the EMP\_TAB table as he or she chooses.

## Limiting Roles Through PRODUCT\_USER\_PROFILE

Oracle8i offers some capability to limit what roles a user accesses through an application, via the PRODUCT\_USER\_PROFILE table.

DBAs can use PRODUCT\_USER\_PROFILE to disable certain SQL and SQL\*Plus commands in the SQL\*Plus environment on a per-user basis. SQL\*Plus—not Oracle—enforces this security. DBAs can even restrict access to the GRANT, REVOKE, and SET ROLE commands in order to control users' ability to change their database privileges.

The PRODUCT\_USER\_PROFILE table enables you to list roles which you do not want users to activate with an application. You can also explicitly disable use of various commands, such as SET ROLE. For example, you could create an entry in the PRODUCT\_USER\_PROFILE table to:

- disallow use of the CLERK and MANAGER roles with SQL\*Plus
- disallow use of SET ROLE with SQL\*Plus

Suppose user Jane connects to the database using SQL\*Plus. Jane has the CLERK, MANAGER, and ANALYST roles. As a result of the above entry in PRODUCT\_USER\_PROFILE, Jane is only able to exercise her ANALYST role with SQL\*Plus. Also, when Jane attempts to issue a SET ROLE statement, she is explicitly prevented from

doing so because of the entry in the `PRODUCT_USER_PROFILE` table prohibiting use of `SET ROLE`.

Use of the `PRODUCT_USER_PROFILE` table does not completely guarantee security, for multiple reasons. In the above example, while `SET ROLE` is disallowed with SQL\*Plus, if Jane had other privileges granted to her directly, she could exercise these using SQL\*Plus.

**See Also:** *SQL\*Plus User's Guide and Reference* for more information about the `PRODUCT_USER_PROFILE` table

### Virtual Private Database for Greatest Security

Oracle8i allows user to enforce security—to a fine level of granularity—directly on tables or views using virtual private database: the combination of server-enforced fine-grained access control and application context. Because security policies are attached directly to tables or views and automatically applied whenever a user access data, there is no way to bypass security.

Strong security policies, centrally managed and applied directly to data, enable security to be enforced no matter how a user gets to the data, whether through an application, by a query, or by using a report-writing tool.

With virtual private database, a user directly or indirectly accessing a table or view having a security policy associated with it, causes the server to dynamically modify the statement based on a `WHERE` condition (known as a predicate) returned by a function which implements the security policy. The user's SQL statement is modified dynamically, transparently to the user, using any condition which can be expressed in, or returned by a function.

Functions which return predicates can also include callouts to other functions. You could embed a C or Java callout within your PL/SQL package that could either access operating system information or return `WHERE` clauses from an operating system file or central policy store. A policy function can return different predicates for each user, each group of users, or each application.

Application context enables you to securely access attributes on which you base your security policies. For example, users with the attribute of `manager` would have a different security policy than users with the attribute of `employee`.

Consider an HR clerk who is only allowed to see employee records in the Aircraft Division. When the user initiates the query

```
SELECT * FROM emp;
```

the function implementing the security policy returns the predicate `division = 'AIRCRAFT'`, and the database transparently rewrites the query. The query actually executed becomes:

```
SELECT * FROM emp WHERE division = 'AIRCRAFT';
```

The security policy is applied within the database itself, rather than an application, which means that use of a different application will not bypass the security policy. Also, security can be built once, in the database, instead of being reimplemented in multiple applications. Virtual private database therefore provides far stronger security than application-based security, at a lower cost of ownership.

**See Also:** ["Ways to Use Application Context with Fine Grained Access Control"](#) on page 11-47

## Recommended Application Design Practices to Reduce Risk

To avoid potential problems, use the following recommended practices when implementing application roles. Each practice is explained in detail below.

- [Enable and Disable Roles Promptly](#)
- [Encapsulate Privileges in Stored Procedures](#)
- [Use Role Passwords Unknown to the User](#)
- [Use Application Context and Fine-Grained Access Control](#)

### Enable and Disable Roles Promptly

Enable the proper role when the application starts, and disable it when the application terminates

1. Give each application distinct roles:

One role should contain *all* privileges necessary to use the application successfully. Depending on the situation, there might be several roles that contain more or fewer privileges to provide tighter or less restrictive security while executing the application. Each application role should be protected by a password (or by operating system authentication) to prevent unauthorized use.

Another role should contain only non-destructive privileges associated with the application (`SELECT` privileges for specific tables or views associated with the application). The read-only role allows the application user to generate custom reports using ad hoc tools, such as SQL\*Plus. However, this role does not allow the application user to modify table data outside the application itself. A role designed for an ad hoc query tool may or may not be protected by a password (or by operating system authentication).

2. At startup, each application should use the `SET ROLE` statement to enable one of the application roles associated with that application. If a password is used to authorize the role, then the password must be included in the `SET ROLE` statement within the application (encrypted by the application, if possible). If the role is authorized by the operating system, then the system administrator must have set up user accounts and applications so that application users get the appropriate operating system privileges when using the application.
3. At termination, each application should disable the previously enabled application role.
4. Application users should be granted application roles, as required.

**Note:** Roles granted to users can nonetheless be enabled by users *outside the application*. Such use is not controlled by application-based security. Again, virtual private database is the best way to solve this problem. See "[Ways to Use Application Context with Fine Grained Access Control](#)" on page 11-47.

Additionally, you can:

5. Specify the roles to enable when a user starts SQL\*Plus, using the `PRODUCT_USER_PROFILE` table. This functionality is similar to that of a precompiler or Oracle Call Interface (OCI) application that issues a `SET ROLE` statement to enable specific roles upon application startup.
6. Disable the use of the `SET ROLE` statement for SQL\*Plus users with the `PRODUCT_USER_PROFILE` table. This allows a SQL\*Plus user only the privileges associated with the roles enabled when the user started SQL\*Plus.

Other ad hoc query and reporting tools can also make use of the `PRODUCT_USER_PROFILE` table to restrict the roles and commands that each user can use while running that product.

**See Also:** The appropriate tool manual, such as the *SQL\*Plus User's Guide and Reference*

### Encapsulate Privileges in Stored Procedures

Another way to restrict users from exercising application privileges by way of ad hoc query tools is to encapsulate privileges into stored procedures. Grant users execute privileges on the procedures, rather than issuing them direct privilege grants. In this way, the logic goes with the privilege.

This allows users to exercise privileges only in the context of well-formed business applications. For example, consider authorizing users to update a table only by executing a stored procedure, rather than by updating the table directly. By doing this, you avoid the problem of the user having the `SELECT` privilege and using it outside the application.

**See Also:** "[Example 3: Human Resources Application #2](#)" on page 11-59

### **Use Role Passwords Unknown to the User**

Grant privileges through roles that require a password unknown to the user.

If there are privileges which the user should use only within the application, you can enable the role by a password *known only by the creator of the role*. Use the application to issue a `SET ROLE` statement. Because the user does not have the password, either embed the password in the application, or use a stored procedure to retrieve the role password from a database table.

This measure discourages users from avoiding use of the application. However, while it does improve application security, it is not foolproof. It still has the following vulnerabilities:

- A user with access to application code could potentially find the password embedded in the application.
- The user would still need `EXECUTE` permission on the stored procedure used to retrieve the password. The user could execute the procedure, retrieve the password, then use the role outside of the application.

### **Use Application Context and Fine-Grained Access Control**

In this scenario, you combine server-enforced fine-grained access control and, through application context, session-based attributes.

**See Also:** ["Ways to Use Application Context with Fine Grained Access Control"](#) on page 11-47

## Protecting Database Objects Through Use of Schemas

A *schema* is a security domain that can contain database objects. The privileges granted to each user or role control access to these database objects. This section covers:

- [Unique Schemas](#)
- [Schema-independent Users](#)

### Unique Schemas

Most schemas can be thought of as usernames: the accounts which enable users to connect to a database and access the database objects. However, *unique schemas* do not allow connections to the database, but are used to contain a related set of objects. Schemas of this sort are created as normal users, yet are not granted the `CREATE SESSION` system privilege (either explicitly or via a role). However, you must temporarily grant the `CREATE SESSION` and `RESOURCE` privilege to such schemas, if you want to use the `CREATE SCHEMA` statement to create multiple tables and views in a single transaction.

For example, the schema objects for a specific application might be owned by a schema. Application users can connect to the database using typical database usernames and use the application and the corresponding objects, if they have the privileges to do so. However, no user can connect to the database using the schema set up for the application. This configuration prevents access to the associated objects via the schema, and provides another layer of protection for schema objects. In this case, the application could issue an `ALTER SESSION SET SCHEMA` statement to connect the user to the correct application schema.

### Schema-independent Users

For many applications, users do not need their own accounts—or their own schemas—in a database. These users merely need to access an application schema. For example, users John, Firuzeh and Jane are all users of the Payroll application, and they need access to the `PAYROLL` schema on the `FINANCE` database. None of them need to create their own objects in the database; in fact, they need only access Payroll objects. To address this issue, Oracle Advanced Security provides enterprise users and schema-independent users.

Enterprise users—users managed in a directory service—can access a shared schema. They do not need to be created as database users; they are schema-independent users of the database. Instead of creating a user account (that is, a user schema) in each database an enterprise user needs to access, as well as

creating the user in the directory, an administrator can create an enterprise user once, in the directory, and point the user at a shared schema that many other enterprise users can also access.

In the previous example, if John, Firuzeh and Jane all access the `Sales` database as well as the `Finance` database, an administrator need only create a single schema in the `Sales` database, which all three users can access—instead of creating an account for each user on the `Sales` database. In this case, the DBA for the `Sales` database creates a shared schema called `sales_application`, as follows:

```
CREATE USER sales_application IDENTIFIED GLOBALLY AS '';
```

The mapping between enterprise users and a schema is done in the directory by means of one or more mapping objects. A mapping object maps the Distinguished Name (DN) of a user, contained in the user's X.509 certificate, to a database schema that the user will access. This can be done in one of two ways:

- A full DN mapping maps the DN of a single directory user to a database schema, thus associating this user with a particular schema on a database.
- A partial DN mapping maps all users who share part of a DN to a database schema. A partial DN mapping is useful if multiple enterprise users that have something in common are already grouped under some common root in the directory tree. For example, all enterprise users in the directory subtree corresponding to the engineering division can be mapped to one shared schema on the bug database. Multiple enterprise users, who share part of their DN, can thus access the same shared schema.

When the database tries to determine the enterprise user's schema in the directory (that is, the schema to which the database will connect the user) it first looks for a full DN mapping. If it does not find a full DN mapping, then it looks for a partial one. A full DN mapping thus takes precedence over a partial one.

If a set of privileges need to be granted to a group of users, this can be done by granting roles and privileges to a shared schema. Every user sharing the schema gets these local roles and local privileges in addition to the enterprise roles.

Each enterprise user can be mapped to a shared schema on each database she needs to access; these schema-independent users thus need not have a dedicated database schema on each database. User-schema separation therefore lowers the cost of managing users in an enterprise.

**See Also:** ["Renaming the Schema"](#) on page 2-40 in [Chapter 2](#), ["Managing Schema Objects"](#)

*Oracle Advanced Security Administrator's Guide*

## Managing Object Privileges

As part of designing your application, you need to determine the types of users who will be working with the application and the level of access they need to accomplish their designated tasks. You must categorize these users into role groups, and then determine the privileges that must be granted to each role.

### Object Privileges

End users are typically granted object privileges. An object privilege allows a user to perform a particular action on a specific table, view, sequence, procedure, function, or package. [Table 11-1](#) summarizes the object privileges available for each type of object.

**Table 11-1 Object Privileges**

Object Privilege	Table	View	Sequence	Procedure <sup>(1)</sup>
ALTER	3		3	
DELETE	3	3		
EXECUTE				3
INDEX	3 (2)			
INSERT	3	3		
REFERENCES	3 (2)			
SELECT	3	3 (3)	3	
UPDATE	3	3		

#### Notes:

- 1 Stand-alone stored procedures, functions, and public package constructs
- 2 Privilege that cannot be granted to a role
- 3 Can also be granted for snapshots

## SQL Statements Permitted by Object Privileges

[Table 11-2](#) lists the SQL statements permitted by the object privileges shown in [Table 11-1](#).

As you implement and test your application, you should create each of these roles. Test the usage scenario for each role to be certain that the users of your application will have proper access to the database. After completing your tests, coordinate with the administrator of the application to ensure that each user is assigned the proper roles.

**Table 11-2** *SQL Statements Permitted by Database Object Privileges*

<b>Object Privilege</b>	<b>SQL Statements Permitted</b>
ALTER	ALTER object (table or sequence) CREATE TRIGGER ON object (tables only)
DELETE	DELETE FROM object (table or view)
EXECUTE	EXECUTE object (procedure or function) References to public package variables
INDEX	CREATE INDEX ON object (table or view)
INSERT	INSERT INTO object (table or view)
REFERENCES	CREATE or ALTER TABLE statement defining a FOREIGN KEY integrity constraint on object (tables only)
SELECT	SELECT...FROM object (table, view, or snapshot) SQL statements using a sequence

## Creating a Role and Protecting Its Use

This section explains how to create a new role and protect its use.

### Creating and Implementing a New Role

To create a role, you must have the `CREATE ROLE` system privilege.

The name of a new role must be unique among existing usernames and role names of the database. Roles are not contained in the schema of any user.

Immediately after creation, a role has no privileges associated with it. To associate privileges with a new role, you must grant it privileges or other roles.

### Managing Roles

You can create roles such that their use is authorized using information from the operating system or from a network authentication service. This enables role management to be centralized.

Central management of roles provides many benefits. If an employee leaves, for example, all of her roles and permissions can be changed in a single place.

### Protecting Role Use

The use of a role can be protected by an associated password. For example:

```
CREATE ROLE Clerk IDENTIFIED BY Bicentennial;
```

A user who is granted a role protected by a password can enable or disable the role only by supplying the proper password for the role using a `SET ROLE` statement. If a role is created without any protection, then any grantee can enable or disable it.

Separate `SET ROLE` statements can be used to enable one application role, and disable all other roles of a user. This way, the user cannot use privileges (from a role) intended for another application. With ad hoc query tools, such as SQL\*Plus or Enterprise Manager, users can explicitly enable only the roles for which they are authorized.

**See Also:** ["Explicitly Enabling Roles"](#) on page 11-25 and ["Restricting Application Roles from Tool Users"](#) on page 11-12

*Oracle8i Administrator's Guide*

For information about network authentication services, see *Oracle Advanced Security Administrator's Guide*.

## Enabling and Disabling Roles

Although a user can be granted a role, the role must be enabled before the privileges associated with it become available in the user's current session. Some, all, or none of the user's roles can be enabled or disabled. The following sections discuss when roles should be enabled and disabled, and the different ways in which a user can have roles enabled or disabled.

### When to Enable Roles

In general, a user's security domain should permit the user to perform the task at hand, yet limit the user from having privileges which are not necessary for the current job. For example, a user should have all the privileges to work with the database application currently in use, but not have any privileges required for any other database applications. Having too many privileges might allow users to access information through unintended methods.

Privileges granted directly to a user are always available to the user; therefore, directly granted privileges cannot be selectively enabled and disabled, depending on a user's current task. By contrast, privileges granted to a role can be selectively made available for any user granted the role. The enabling of roles *never* affects privileges explicitly granted to a user. The following sections explain how a user's roles can be selectively enabled (and disabled).

### Default Roles

A default role is automatically enabled for a user when the user creates a session. A user's list of default roles should include those which correspond to his or her typical job function.

Each user has a list of zero, one, or more default roles. Any role directly granted to a user can potentially be a default role of the user. An indirectly granted role (a role that is granted to a role) cannot be a default role.

The number of default roles for a user should not exceed the maximum number of enabled roles that are allowed per user (as specified by the initialization parameter `MAX_ENABLED_ROLES`); if the number of default roles for a user exceeds this maximum, then errors are returned when the user attempts a connection, and the user's connection is not allowed.

---

---

**Note:** A default role is automatically enabled for a user when the user creates a session. Placing a role in a user's list of default roles bypasses authentication for the role, whether it is authorized using a password or the operating system.

---

---

A user's list of default roles can be set and altered using the SQL statement `ALTER USER`. If the user's list of default roles is specified as `ALL`, then every role granted to a user is automatically added to the user's list of default roles. Only subsequent modification of a user's default role list can remove newly granted roles from a user's list of default roles.

Modifications to a user's default role list only apply to sessions created after the alteration or role grant; neither method applies to a session in progress at the time of the user alteration or role grant.

### Explicitly Enabling Roles

Any user (or application) can use the `SET ROLE` statement to enable any granted roles, provided the grantee supplies role passwords, when necessary.

A `SET ROLE` statement enables all specified roles, provided that they have been granted to the user. All roles granted to the user that are not explicitly specified in a `SET ROLE` statement are disabled, including any roles previously enabled.

When you enable a role that contains other roles, all the indirectly granted roles are specifically enabled. Each indirectly granted role can be explicitly enabled or disabled for a user.

If a role is protected by a password, then the role can only be enabled by indicating the role's password in the `SET ROLE` statement. If the role is not protected by a password, then the role can be enabled with a simple `SET ROLE` statement. For example, assume that Morris' security domain is as follows:

- He is granted three roles:
  - `PAYROLL_CLERK` (password `BICENTENNIAL`)
  - `ACCTS_PAY` (password `GARFIELD`)
  - `ACCTS_REC` (identified externally).

The `PAYROLL_CLERK` role includes the indirectly granted role `PAYROLL_REPORT` (identified externally).
- His only default role is `PAYROLL_CLERK`.

Morris' currently enabled roles can be changed from his default role, `PAYROLL_CLERK`, to `ACCTS_PAY` and `ACCTS_REC`, by the following statements:

---

---

**Note:** You may need to set up the following data structures for certain examples to work, such as:

```
GRANT PAYROLL_CLERK TO hr;  
GRANT ACCTS_PAY TO hr;  
GRANT ACCTS_REC TO hr;
```

---

---

```
SET ROLE accts_pay IDENTIFIED BY garfield;  
SET ROLE accts_pay IDENTIFIED BY accts_rec;
```

Notice that in the first statement, multiple roles can be enabled in a single `SET ROLE` statement. The `ALL` and `ALL EXCEPT` options of the `SET ROLE` statement also allow several roles granted directly to the user to be enabled in one statement:

---

---

**Note:** You may need to set up the following data structures for certain examples to work, such as:

```
CREATE ROLE Payroll_clerk;  
CREATE ROLE Payroll_report;
```

---

---

```
SET ROLE ALL EXCEPT Payroll_clerk;
```

This statement shows the use of the `ALL EXCEPT` option of the `SET ROLE` statement. Use this option when you want to enable most of a user's roles and only disable one or more. Similarly, all of Morris' roles can be enabled by the following statement:

```
SET ROLE ALL;
```

When using the `ALL` or `ALL EXCEPT` options of the `SET ROLE` statement, all roles to be enabled either must not require a password, or must be authenticated using the operating system. If a role requires a password, then the `SET ROLE ALL` or `ALL EXCEPT` statement is rolled back and an error is returned.

A user can also explicitly enable any indirectly granted roles granted to him or her via an explicit grant of another role. For example, Morris can issue the following statement:

```
SET ROLE Payroll_report;
```

## Enabling and Disabling Roles When OS\_ROLES=TRUE

If `OS_ROLES` is set to `TRUE`, then any role granted by the operating system can be dynamically enabled using the `SET ROLE` statement. However, any role not identified in a user's operating system account cannot be specified in a `SET ROLE` statement (it is ignored), even if a role has been granted using a `GRANT` statement.

When `OS_ROLES` is set to `TRUE`, a user can enable as many roles as specified by the initialization parameter `MAX_ENABLED_ROLES`.

**See Also:** *Oracle8i Administrator's Guide* for more information about use of the operating system for role authorization.

## Dropping Roles

When you drop a role, the security domains of all users and roles granted that role are immediately changed to reflect the absence of the dropped role's privileges. All indirectly granted roles of the dropped role are also removed from affected security domains. Dropping a role automatically removes the role from all users' default role lists.

Because the creation of objects is not dependent upon the privileges received via a role, no cascading effects regarding objects need to be considered when dropping a role (for example, tables or other objects are not dropped when a role is dropped).

Drop a role using the SQL statement `DROP ROLE`. For example:

```
DROP ROLE clerk;
```

To drop a role, you must have the `DROP ANY ROLE` system privilege or have been granted the role with the `ADMIN OPTION`.

## Granting and Revoking System Privileges and Roles

The following sections explain how to grant and revoke system privileges and roles.

### Granting System Privileges and Roles

System privileges and roles can be granted to other roles or users using the SQL command `GRANT`, as shown in the following example:

---

---

**Note:** You may need to set up the following data structures for certain examples to work:

```
CONNECT sys/change_on_install AS sysdba;
CREATE USER jward IDENTIFIED BY jward;
CREATE USER tsmith IDENTIFIED BY tsmith;
CREATE USER finance IDENTIFIED BY finance;
CREATE USER michael IDENTIFIED BY michael;
CREATE ROLE Payroll_report;
GRANT CREATE TABLE, Accts_rec TO finance IDENTIFIED BY finance;
GRANT CREATE TABLE, Accts_rec TO tsmith IDENTIFIED BY tsmith;
GRANT REFERENCES ON Dept_tab TO jward;
CONNECT scott/tiger
CREATE VIEW Salary AS SELECT Empno,Sal from Emp_tab;
```

---

---

```
GRANT CREATE SESSION, Accts_pay TO jward, finance;
```

Schema object privileges cannot be granted along with system privileges and roles in the same `GRANT` statement.

### Granting System Privileges and Roles with the ADMIN OPTION

A system privilege or role can be granted with the `ADMIN OPTION`. (This option is not valid when granting a role to another role.) A grantee with this option has several expanded capabilities:

- The grantee can grant or revoke the system privilege or role to or from *any* user or other role in the database. (A user cannot revoke a role from himself.)
- The grantee can further grant the system privilege or role with the `ADMIN OPTION`.
- The grantee of a role can alter or drop the role.

A grantee without the `ADMIN OPTION` cannot perform the above operations.

When a user creates a role, the role is automatically granted to the creator with the `ADMIN OPTION`.

Assume that you grant the `NEW_DBA` role to `MICHAEL` with the following statement:

```
GRANT New_dba TO michael WITH ADMIN OPTION;
```

The user `MICHAEL` cannot only use all of the privileges implicit in the `NEW_DBA` role, but can grant, revoke, or drop the `NEW_DBA` role, as necessary.

**Privileges Required to Grant System Privileges or Roles** To grant a system privilege or role, the grantor requires the `ADMIN OPTION` for all system privileges and roles being granted. Additionally, any user with the `GRANT ANY ROLE` system privilege can grant any role in a database.

## Revoking System Privileges and Roles

System privileges and roles can be revoked using the SQL command `REVOKE`. For example:

```
REVOKE CREATE TABLE, Accts_rec FROM tsmith, finance;
```

The `ADMIN OPTION` for a system privilege or role cannot be selectively revoked; the privilege or role must be revoked, and then the privilege or role is regranted without the `ADMIN OPTION`.

**Privileges Required to Revoke System Privileges and Roles** Any user with the `ADMIN OPTION` for a system privilege or role can revoke the privilege or role from any other database user or role (the user does not have to be the user that originally granted the privilege or role). Additionally, any user with the `GRANT ANY ROLE` can revoke *any* role.

**Cascading Effects of Revoking System Privileges** There are no cascading effects when revoking a system privilege related to DDL operations, regardless of whether the privilege was granted with or without the `ADMIN OPTION`. For example, assume the following:

1. You grant the `CREATE TABLE` system privilege to `JWARD` with the `WITH ADMIN OPTION`.
2. `JWARD` creates a table.
3. `JWARD` grants the `CREATE TABLE` system privilege to `TSMITH`.
4. `TSMITH` creates a table.

5. You revoke the `CREATE TABLE` privilege from `JWARD`.
6. `JWARD`'s table continues to exist. `TSMITH` continues to have the `CREATE TABLE` system privilege, and his table still exists.

Cascading effects can be observed when revoking a system privilege related to a DML operation. For example, if `SELECT ANY TABLE` is granted to a user, and if that user has created any procedures, then all procedures contained in the user's schema must be reauthorized before they can be used again (after the revoke).

## Granting and Revoking Schema Object Privileges and Roles

Grant schema object privileges to roles or users using the SQL command `GRANT`. The following statement grants the `SELECT`, `INSERT`, and `DELETE` object privileges for all columns of the `EMP_TAB` table to the users `JWARD` and `TSMITH`:

```
GRANT SELECT, INSERT, DELETE ON Emp_tab TO jward, tsmith;
```

To grant the `INSERT` object privilege for only the `ENAME` and `JOB` columns of the `EMP_TAB` table to the users `JWARD` and `TSMITH`, enter the following statement:

```
GRANT INSERT(Ename, Job) ON Emp_tab TO jward, tsmith;
```

To grant all schema object privileges on the `SALARY` view to the user `WALLEN`, use the `ALL` short cut. For example:

```
GRANT ALL ON Salary TO wallen;
```

System privileges and roles cannot be granted along with schema object privileges in the same `GRANT` statement.

## Granting and Revoking Schema Object Privileges with the GRANT OPTION

A schema object privilege can be granted to a user with the `GRANT OPTION`. This special privilege allows the grantee several expanded privileges:

- The grantee can grant the schema object privilege to any user or any role in the database.
- The grantee can also grant the schema object privilege to other users, with or without the `GRANT OPTION`.
- If the grantee receives schema object privileges for a table with the `GRANT OPTION`, and the grantee has the `CREATE VIEW` or the `CREATE ANY VIEW` system privilege, then the grantee can create views on the table and grant the corresponding privileges on the view to any user or role in the database.

The user whose schema contains an object is automatically granted all associated schema object privileges with the `GRANT OPTION`.

---

---

**Note:** The `GRANT OPTION` is not valid when granting a schema object privilege to a role. Oracle prevents the propagation of schema object privileges via roles, so that grantees of a role cannot propagate object privileges received via roles.

---

---

**Privileges Required to Grant Schema Object Privileges** To grant a schema object privilege, the grantor must either

- Be the owner of the schema object being specified, or
- Have been granted the schema object privileges being granted with the `GRANT OPTION`

### Revoking Schema Object Privileges

Schema object privileges can be revoked using the SQL command `REVOKE`. For example, assuming you are the original grantor, to revoke the `SELECT` and `INSERT` privileges on the `EMP_TAB` table from the users `JWARD` and `TSMITH`, enter the following statement:

```
REVOKE SELECT, INSERT ON Emp_tab FROM jward, tsmith;
```

A grantor could also revoke all privileges on the table `DEPT_TAB` (even if only one privilege was granted) that he or she granted to the role `HUMAN_RESOURCES` by entering the following statement:

```
REVOKE ALL ON Dept_tab FROM human_resources;
```

This statement would only revoke the privileges that the grantor authorized, not the grants made by other users. The `GRANT OPTION` for a schema object privilege cannot be selectively revoked; the schema object privilege must be revoked and then regranted without the `GRANT OPTION`. A user cannot revoke schema object privileges from him or herself.

**Revoking Column-Selective Schema Object Privileges** Recall that column-specific `INSERT`, `UPDATE`, and `REFERENCES` privileges can be granted for tables or views; however, it is not possible to revoke column-specific privileges selectively with a similar `REVOKE` statement. Instead, the grantor must first revoke the schema object privilege for all columns of a table or view, and then selectively grant the new column-specific privileges again.

For example, assume the role `HUMAN_RESOURCES` has been granted the `UPDATE` privilege on the `DEPTNO` and `DNAME` columns of the table `DEPT_TAB`. To revoke the `UPDATE` privilege on just the `DEPTNO` column, enter the following two statements:

```
REVOKE UPDATE ON Dept_tab FROM human_resources;  
GRANT UPDATE (Dname) ON Dept_tab TO human_resources;
```

The `REVOKE` statement revokes the `UPDATE` privilege on all columns of the `DEPT_TAB` table from the role `HUMAN_RESOURCES`. The `GRANT` statement regrants the `UPDATE` privilege on the `DNAME` column to the role `HUMAN_RESOURCES`.

**Revoking the REFERENCES Schema Object Privilege** If the grantee of the `REFERENCES` object privilege has used the privilege to create a foreign key constraint (that currently exists), then the grantor can only revoke the privilege by specifying the `CASCADE CONSTRAINTS` option in the `REVOKE` statement:

```
REVOKE REFERENCES ON Dept_tab FROM jward CASCADE CONSTRAINTS;
```

Any foreign key constraints currently defined that use the revoked `REFERENCES` privilege are dropped when the `CASCADE CONSTRAINTS` option is specified.

**Privileges Required to Revoke Schema Object Privileges** To revoke a schema object privilege, the revoker must be the original grantor of the object privilege being revoked.

**Cascading Effects of Revoking Schema Object Privileges** Revoking a schema object privilege can have several types of cascading effects that should be investigated before a `REVOKE` statement is issued:

- Schema object definitions that depend on a DML object privilege can be affected if the DML object privilege is revoked. For example, assume the procedure body of the `TEST` procedure includes a SQL statement that queries data from the `EMP_TAB` table. If the `SELECT` privilege on the `EMP_TAB` table is revoked from the owner of the `TEST` procedure, then the procedure can no longer be executed successfully.
- Schema object definitions that require the `ALTER` and `INDEX` DDL object privileges are not affected, if the `ALTER` or `INDEX` object privilege is revoked. For example, if the `INDEX` privilege is revoked from a user that created an index on someone else's table, then the index continues to exist after the privilege is revoked.
- When a `REFERENCES` privilege for a table is revoked from a user, any foreign key integrity constraints defined by the user that require the dropped

REFERENCES privilege are automatically dropped. For example, assume that the user JWARD is granted the REFERENCES privilege for the DEPTNO column of the DEPT\_TAB table and creates a foreign key on the DEPTNO column in the EMP\_TAB table that references the DEPTNO column. If the REFERENCES privilege on the DEPTNO column of the DEPT\_TAB table is revoked, then the foreign key constraint on the DEPTNO column of the EMP\_TAB table is dropped in the same operation.

- The schema object privilege grants propagated using the GRANT OPTION are revoked, if a grantor's object privilege is revoked. For example, assume that USER1 is granted the SELECT object privilege with the GRANT OPTION, and grants the SELECT privilege on EMP\_TAB to USER2. Subsequently, the SELECT privilege is revoked from USER1. This revoke is cascaded to USER2 as well. Any schema objects that depended on USER1's and USER2's revoked SELECT privilege can also be affected.

**How Grants Affect Dependent Objects** Issuing a GRANT statement against a schema object causes the "last DDL time" attribute of the object to change. This can invalidate any dependent schema objects, in particular PL/SQL package bodies that refer to the schema object. These then must be recompiled.

## Granting to, and Revoking from, the User Group PUBLIC

Privileges and roles can also be granted to and revoked from the user group `PUBLIC`. Because `PUBLIC` is accessible to every database user, all privileges and roles granted to `PUBLIC` are accessible to every database user.

You should only grant a privilege or role to `PUBLIC` if every database user requires the privilege or role. This recommendation restates the general rule that at any given time, each database user should only have the privileges required to successfully accomplish the current task.

Revokes from `PUBLIC` can cause significant cascading effects, depending on the privilege that is revoked. If any privilege related to a DML operation is revoked from `PUBLIC` (for example, `SELECT ANY TABLE`, `UPDATE ON EMP_TAB`), then all procedures in the database (including functions and packages) must be reauthorized before they can be used again. Therefore, use caution when granting DML-related privileges to `PUBLIC`.

**When Grants and Revokes Take Effect** Depending upon what is granted or revoked, a grant or revoke takes effect at different times:

- All grants/revokes of privileges (system and schema object) to users, roles, or `PUBLIC` are immediately observed.
- All grants/revokes of roles to users, other roles, or `PUBLIC` are observed only when a current user session issues a `SET ROLE` statement to re-enable the role after the grant/revoke, or when a new user session is created after the grant/revoke.

## Fine-Grained Access Control

Fine-grained access control allows you to build applications that enforce security policies at a low level of granularity. You can use it, for example, to restrict a customer who is accessing an Oracle server to see only his own account, a physician to see only the records of her own patients, or a manager to see only the records of employees who work for him.

When you use fine-grained access control, you create security policy functions attached to the table or view on which you have based your application. Then, when a user enters a DML statement (`SELECT`, `INSERT`, `UPDATE`, or `DELETE`) on that object, Oracle dynamically modifies the user's statement—transparently to the user—so that the statement implements the correct access control.

This section covers:

- [Features of Fine-Grained Access Control](#)
- [How to Add a Policy to a Table or View](#)
- [Example of a Dynamically Modified Statement](#)

## Features of Fine-Grained Access Control

Fine-grained access control provides the following capabilities:

- [Table- Or View-based Security Policies](#)
- [Multiple Policies for Each Table or View](#)
- [High Performance](#)

### Table- Or View-based Security Policies

Attaching security policies to tables or views, rather than to applications, provides greater security, simplicity, and flexibility.

Security	Attaching a policy to a table or view overcomes a potentially serious application security problem. Suppose a user is authorized to use an application, and then, drawing on the privileges associated with that application, wrongfully modifies the database by using an ad hoc query tool, such as SQL*Plus. By attaching security policies to tables or views, fine-grained access control ensures that the same security is in force, no matter how a user accesses the data.
Simplicity	Adding the security policy to the table or view means that you make the addition only once, rather than repeatedly adding it to each of your table- or view-based applications.
Flexibility	You can have one security policy for <code>SELECT</code> statements, another for <code>INSERT</code> statements, and still others for <code>UPDATE</code> and <code>DELETE</code> statements. For example, you might want to enable a Human Resources clerk to <code>SELECT</code> all employee records in her division, but to <code>UPDATE</code> only salaries for those employees in her division whose last names begin with "A" through "F."

### Multiple Policies for Each Table or View

You can establish several policies for the same table or view. For example, suppose you have a base application for Order Entry, and each division of your company has its own special rules for data access. You can add a division-specific policy function to a table without having to rewrite the policy function of the base application.

Note that all policies applied to a table are enforced with `AND` syntax. Thus, if you have three policies applied to the `CUSTOMERS` table, each of these policies is applied to any access of the table. You cannot partition the policies by application.

### High Performance

With fine-grained access control, each policy function for a given query is evaluated only once, at statement parse time. Also, the entire dynamically modified query is optimized and the parsed statement can be shared and reused. This means that rewritten queries can take advantage of the high performance features of Oracle such as dictionary caching and shared cursors.

---

---

**Note:** For performance reasons, parsed static SQL statements in instantiated PL/SQL packages may not get re-parsed. If you want to force the re-evaluation of the policy function, Oracle recommends that you use dynamic SQL. For more information, see ["Dynamic SQL Recommended for Use with SYS\\_CONTEXT"](#) on page 11-50.

---

---

## How to Add a Policy to a Table or View

The `DBMS_RLS` package enables you to administer security policies. This package includes four procedures:

**Table 11–3** *DBMS\_RLS Procedures*

Procedure	Purpose
<code>DBMS_RLS.ADD_POLICY</code>	Use this procedure to add a policy to a table or view.
<code>DBMS_RLS.DROP_POLICY</code>	Use this procedure to drop a policy from a table or view.
<code>DBMS_RLS.REFRESH_POLICY</code>	Use this procedure to force a reparse of open cursors associated with a policy, so that a new policy or change to a policy can take effect immediately.
<code>DBMS_RLS.ENABLE_POLICY</code>	To enable (or disable) a policy you have previously added to a table or view.

These procedures allow you to specify the table or view to which you are adding a policy, the name of the policy, the function which implements the policy, the type of statement to which the policy applies (that is, `SELECT`, `INSERT`, `UPDATE`, or `DELETE`), and additional information.

For example, the `ADD_POLICY` procedure includes an `UPDATE_CHECK` parameter to check the policy before and after insert (or update). If you cannot see it after update or insert, then the update or insert is not allowed. Oracle recommends that you use the `UPDATE_CHECK` parameter to spare users the frustration of being able to insert records which they cannot later select.

**See Also:** *Oracle8i Supplied PL/SQL Packages Reference*

## Example of a Dynamically Modified Statement

Suppose you want to attach to the `ORDERS_TAB` table the following security policy: "Customers can see only their own orders." The process would be as follows.

1. You create a function to add a predicate to a user's DML statement.

---



---

**Note:** A predicate is the `WHERE` clause and, more explicitly, a selection criteria clause based on one of the operators (`=`, `!=`, `IS`, `IS NOT`, `>`, `>=`).

---



---

In this case, you might create a function that adds the following predicate:

```
Cust_no = (SELECT Custno FROM Customers WHERE Custname =
          SYS_CONTEXT ('userenv', 'session_user'))
```

2. A user enters the statement:

```
SELECT * FROM Orders_tab
```

3. The Oracle server calls the function you created to implement the security policy.
4. The function dynamically modifies the user's statement to read:

```
SELECT * FROM Orders_tab WHERE Custno = (
  SELECT Custno FROM Customers
    WHERE Custname = SYS_CONTEXT('userenv', 'session_user'))
```

5. The Oracle server executes the dynamically modified statement.
6. Upon execution, the function uses the username returned by `SYS_CONTEXT ('userenv', 'session_user')` to look up the corresponding customer and to limit the data returned from the `ORDERS_TAB` table to that customer's data only.

**See Also:** For more information on using fine-grained access control, see "[Examples](#)" on page 11-53 and *Oracle8i Supplied PL/SQL Packages Reference*.

## Application Context

Application context allows you to write applications which draw upon certain aspects of a user's session information. This is especially useful in developing secure applications based on a user's access privileges. It provides a way to define, set, and access attributes that an application can use to enforce access control—specifically, fine-grained access control.

Most applications contain information about the basis on which access is to be limited. In an order entry application, for example, customers would be limited to access their own orders (`ORDER_NUMBER`) and customer number (`CUSTOMER_NUMBER`). These can be used as security attributes.

Consider a user running the Oracle Human Resource application. Part of the application's initialization process is to determine the kind of responsibility that the user can assume, based on the user's identity. This responsibility ID becomes part of the Oracle Human Resource application context; it will affect what data the user can access throughout the session.

This section explains the use of application context. It includes:

- [Features of Application Context](#)
- [Feature Design Principles for Application Context](#)
- [Ways to Use Application Context with Fine Grained Access Control](#)
- [How to Use Application Context](#)

## Features of Application Context

Application context provides important security features:

- [Security Tailored to the Attributes You Specify for Each Application](#)
- [Security Through Validation](#)
- [USERENV Application Context Namespace for Access to Predefined Attributes](#)

### Security Tailored to the Attributes You Specify for Each Application

Each application can have its own context with its own attributes. For example, suppose you have three applications: General Ledger, Order Entry, and Human Resources. You can specify different attributes for each application. Thus,

- For the General Ledger application context, you can specify the attributes `SET_OF_BOOKS` and `TITLE`.
- For the Order Entry application context, you can specify the attribute `CUSTOMER_NUMBER`.
- For the Human Resources application context, you can specify the attributes `ORGANIZATION_ID`, `POSITION`, and `COUNTRY`.

In each case, you can adapt the application context to your precise security needs.

### Security Through Validation

Suppose you have a General Ledger application, which has access control based on set of books. If a user accessing this application changes the set of books he is working on from *01* to *02*, the application context can ensure that:

- Set *02* is a valid set of books.
- The user has privileges to access set of books *02*.

The validation function can check application metadata tables to make this determination and ensure that the attributes in combination are in line with the overall security policy. To restrict a user from changing a context attribute without the above security validation, Oracle verifies that only the designated package which implements the context, changes the attribute.

## USERENV Application Context Namespace for Access to Predefined Attributes

Oracle8i provides a built-in application context namespace, `USERENV`, which provides access to predefined attributes (session primitives—information which the database captures regarding a user's session). For example, the IP address from which a user connected, the username, and a proxy username (in cases where a user connection is proxied through a middle tier), are all available as predefined attributes through the `USERENV` application context.

Predefined attributes can be very useful for access control. For example, if you are using a three-tier application which creates lightweight user sessions through OCI, you can access the `PROXY_USER` attribute in the `USERENV` application context to determine whether the user's session was created by a middle tier application. Your policy function could allow a user to access data only for connections where the user is proxied. If not (that is, in cases where the user is connecting directly to the database), the user would not be able to access any data.

Predefined attributes can be accessed through the `USERENV` application context, but cannot not be changed. They are listed in [Table 11-4](#); see `SYS_CONTEXT` in the *Oracle8i SQL Reference* for complete details about the `USERENV` namespace and its predefined attributes.

Use the following syntax to return information about the current session.

```
SYS_CONTEXT('userenv', 'attribute')
```

**Note:** The `USERENV` application context namespace is intended to replace the `USERENV` function provided in earlier database releases.

**Table 11–4 Key to Predefined Attributes in USERENV Namespace**

Predefined Attribute	Meaning
TERMINAL	Returns the operating system identifier for the client of the current session. "Virtual" in TCP/IP.
LANGUAGE	Returns the language and territory currently used by the session, along with the database character set in the form: <i>language_territory.characterset</i> .
LANG	Returns abbreviation for the language name.
SESSIONID	Returns auditing session identifier.
INSTANCE	Returns the instance identification number of the current instance.
ENTRYID	Returns available auditing entry identifier.
ISDBA	Returns TRUE if you currently have the DBA role enabled and FALSE if you do not.
CLIENT_INFO	Returns up to 64 bytes of user session information that can be stored by an application using the DBMS_APPLICATION_INFO package.
NLS_TERRITORY	Returns the territory of the current session.
NLS_CURRENCY	Returns the currency symbol of the current session.
NLS_CALENDAR	Returns the NLS calendar used for dates in the current session.
NLS_DATE_FORMAT	Returns the current date format of the current session.
NLS_DATE_LANGUAGE	Returns the language used for expressing dates in the current session.
NLS_SORT	Indicates whether the sort base is binary or linguistic.
CURRENT_USER	Returns the name of the user whose privilege the current session is under. Can be different from SESSION_USER from within a stored procedure (such as an invoker-rights procedure).
CURRENT_USERID	Returns the user ID of the user whose privilege the current session is under. Can be different from SESSION_USERID from within a stored procedure (such as an invoker-rights procedure).
SESSION_USER	Returns the database user name by which the current user is authenticated.

**Table 11–4 Key to Predefined Attributes in USERENV Namespace (Cont.)**

<b>Predefined Attribute</b>	<b>Meaning</b>
SESSION_USERID	Returns the identifier of the database user name by which the current user is authenticated.
CURRENT_SCHEMA	Returns the name of the default schema being used in the current session. This can be changed with an ALTER SESSION SET SCHEMA statement.
CURRENT_SCHEMAID	Returns the identifier of the default schema being used in the current session. This can be changed with an ALTER SESSION SET SCHEMAID statement.
PROXY_USER	Returns the name of the database user (typically middle tier) who opened the current session on behalf of SESSION_USER.
PROXY_USERID	Returns the identifier of the database user (typically middle tier) who opened the current session on behalf of SESSION_USER.
DB_DOMAIN	Returns the domain of the database as specified in the DB_DOMAIN initialization parameter.
DB_NAME	Returns the name of the database as specified in the DB_NAME initialization parameter.
HOST	Returns the name for the host machine on which the database is running.
OS_USER	Returns the operating system username of the client process that initiated the database session.
EXTERNAL_NAME	Returns the external name of the database user.
IP_ADDRESS	Returns the IP address of the machine from which the client is connected.
NETWORK_PROTOCOL	Returns the protocol named in the connect string (PROTOCOL= <i>protocol</i> ).
BG_JOB_ID	Returns the background job ID.
FG_JOB_ID	Returns the foreground job ID.
AUTHENTICATION_TYPE	Shows how the user was authenticated (DATABASE, OS, NETWORK, PROXY).
AUTHENTICATION_DATA	Returns the data being used to authenticate the login user. (Returns the certificate content, if one exists.)

## Feature Design Principles for Application Context

Fine-grained access control with application context was designed to handle static applications; that is, those for which the security attributes contained within application contexts are static within the user sessions. In other words, the feature is best used for applications in which the user logs in, his application context is set for the session, and the context does not change until the user logs off. This design principle enables application context to be highly scalable, because many users can share the same fully-parsed, optimized statement.

If an application needs to change a context attribute within a session, then you may need to programmatically ensure that new application context attributes are reflected in the cursor which the user executes. For example, suppose an application wishes to change a user's `position` attribute within the Human Resources context to allow the user to view more data within her session. The application may need to force a cursor reparse to ensure that the new `position` attribute is used to limit access to data.

There are at least three ways to force a cursor reparse:

- [Refresh the Policy](#)
- [Force Explicit Reparse](#)
- [Use Dynamic SQL](#)

### **Refresh the Policy**

The administrative interface to fine-grained access control is the `DBMS_RLS` package. You can use the `DBMS_RLS.REFRESH_POLICY` procedure to refresh the security policy. This will force a commit of the current transaction. A drawback of this approach is that it will flush all cursors associated with the specified table or view, including cursors of users whose application context has not changed. This approach will achieve the desired affect, but it may degrade performance.

**See Also:** *Oracle8i Supplied PL/SQL Packages Reference*

### **Force Explicit Reparse**

Force an explicit reparse of the cursor. For applications using the Oracle Call Interface, you can explicitly reparse an open cursor by issuing a `REPARSE` statement. This forces a SQL statement to be reparsed, which then picks up any change in a users' application context and applies it to the (newly-parsed) statement. PL/SQL, however, does not allow an explicit reparse of a statement, so this approach will not work for PL/SQL statements.

### **Use Dynamic SQL**

For PL/SQL statements, you can use dynamic SQL to create a package associated with a unique context for your application. This approach is described in the following section.

## Ways to Use Application Context with Fine Grained Access Control

To make the implementation of a security policy easier, you have the option of using application context within a fine-grained access control function. Virtual private database (VPD) is the term used for a combination of fine-grained access control with application context.

Application context can be used in very specific ways with fine-grained access control:

- [Using Application Context as a Secure Data Cache](#)
- [Using Application Context to Return a Specific Predicate \(Security Policy\)](#)
- [Using Application Context to Provide Attributes as Bind Variables in a Predicate](#)

### Using Application Context as a Secure Data Cache

Accessing an application context inside your fine-grained access control policy function is like writing down an often-used phone number and posting it next to your phone, where you can find it easily, rather than looking it up every time you need it.

For example, suppose you base access to the `ORDERS_TAB` table on customer number. Rather than querying the customer number for a logged-in user each time you need it, you could store it in the application context. This way, the customer number is available when you need it.

Application context is especially helpful if your security policy is based on multiple security attributes. For example, a policy function which bases a predicate on four attributes (such as employee number, cost center, position, spending limit), would have to execute multiple subqueries to retrieve this information. If all of this data is already available through application context, then performance will be much faster.

## Using Application Context to Return a Specific Predicate (Security Policy)

You can use application context to return the correct predicate—that is, the correct security policy.

An order entry application enforces the policy "customers only see their own orders and clerks see all orders for all customers." Here you have two different policies. You could define an application context with a `position` attribute, and this attribute could be accessed within the policy function to return the correct predicate, depending on the value of the attribute. Thus, you can enable a user in the `Clerk` position to retrieve all orders, but a user in the `Customer` position to see his own records only.

To design a fine-grained access control policy to return a specific predicate for an attribute, access the application context within the function that implements the policy. For example, to limit customers to seeing their own records only, use fine-grained access control to dynamically modify the user's query from this:

```
SELECT * FROM Orders_tab
```

to this:

```
SELECT * FROM Orders_tab
  WHERE Custno = SYS_CONTEXT ('order_entry', 'cust_num');
```

## Using Application Context to Provide Attributes as Bind Variables in a Predicate

Continuing with the example above, suppose you have 50,000 customers, and you do not want to have a different predicate returned for each customer. Customers all share the same policy (that is, they can only see their own orders); it is merely their customer numbers which are different.

Using application context, you can return one predicate within a policy function which applies to 50,000 customers. As a result, there is one shared cursor which nonetheless executes differently for each customer, because the bind variable (the customer number) is evaluated at execution time. This variable is, of course, different for every customer. Use of application context in this case provides optimum performance, as well as fine-grained security.

## How to Use Application Context

To use application context, you perform the following tasks:

- [Task 1: Create a PL/SQL Package that Sets the Context for Your Application](#)
- [Task 2: Create a Unique Context and Associate It with the PL/SQL Package](#)
- [Task 3: Set the Context Before the User Retrieves Data](#)
- [Task 4. Use the Context in a Policy Function](#)

### Task 1: Create a PL/SQL Package that Sets the Context for Your Application

Begin by creating a PL/SQL package with functions that set the context for your application. This section presents an example, followed by a discussion of SQL\_CONTEXT syntax and behavior.

---



---

**Note:** A login trigger is highly recommended because the user's context (information such as EMPNO, GROUP, MANAGER) should be set before the user accesses any data.

---



---

### Example

The following example creates the package `app_security_context`.

```
CREATE OR REPLACE PACKAGE App_security_context IS
    PROCEDURE Set_empno;
END;

CREATE OR REPLACE PACKAGE BODY App_security_context IS
    PROCEDURE Set_empno
    IS
        Emp_id NUMBER;
    BEGIN
        SELECT Empno INTO Emp_id FROM Emp_tab
           WHERE Ename = SYS_CONTEXT('USERENV',
                                     'SESSION_USER');
        DBMS_SESSION.SET_CONTEXT('app_context', 'empno', Emp_id);
    END;
END;
```

**See Also:** *Oracle8i Supplied PL/SQL Packages Reference*

### **SYS\_CONTEXT Syntax**

The syntax for this function is:

```
SYS_CONTEXT ('namespace', 'attribute', [length])
```

It returns the value of `attribute` as defined in the package currently associated with the context namespace. It is evaluated once for each statement execution, and is treated like a constant during type checking for optimization. You can use the pre-defined namespace `USERENV` to access primitive contexts such as `userid` and NLS parameters.

**See Also:** ["USERENV Application Context Namespace for Access to Predefined Attributes"](#) on page 11-42. Also see *Oracle8i SQL Reference* for details about attributes.

### **Dynamic SQL Recommended for Use with SYS\_CONTEXT**

During a session in which you expect a change in policy between executions of a given query, that query must use dynamic SQL.

This is because static SQL and dynamic SQL parse statements differently. With static SQL, statements are parsed at compile time; for performance reasons, they are not reparsed at execution. With dynamic SQL, by contrast, statements are parsed every time they are executed.

Consider a situation in which policy A is in force when you compile a SQL statement—and then you switch to policy B and execute the statement. With static SQL, policy A remains in force: the statement is parsed at compile time and not reparsed upon execution. With dynamic SQL, however, the statement is parsed upon execution, and so the switch to policy B is carried through.

For example, consider the following policy:

```
EMPLOYEE_NAME = SYS_CONTEXT ('userenv', 'session_user')
```

The policy "Employee name matches database user name" is represented in the form of a SQL predicate: the predicate is basically a policy. If the predicate changes, the statement must be reparsed in order to produce the correct result.

### Parallel Query Not Recommended for Use with SYS\_CONTEXT

If `SYS_CONTEXT` is used inside a SQL function which is embedded in a parallel query, the function will not be able to pick up the application context. The application context exists only in the user session.

Consider, for example, a user-defined function within a SQL statement, which sets the user's ID to 5:

```
CREATE FUNC procl AS RETURN NUMBER;
BEGIN
    IF SYS_CONTEXT ('hr', 'id') = 5
    THEN RETURN 1; ELSE RETURN 2;
    END
END;
```

Now consider the statement:

```
SELECT * FROM EMP WHERE procl( ) = 1;
```

If this statement is run as a single query (that is, if one process is used to run the entire query), there will be no problem.

However, if this statement is run as a parallel query, the parallel execution servers (query slave processes) do not have access to the user session which contains the application context information. The query will not produce the expected results.

By contrast, if you use the `SYS_CONTEXT` function within a query, there is no problem. For example,

```
SELECT * FROM EMP WHERE SYS_CONTEXT ('hr', 'id') = 5
```

In this way, it works like a bind variable: the query coordinator can access the application context information and pass it on to the parallel execution servers.

### Versioning in Application Context

When you execute a statement, Oracle8i takes a snapshot of the entire application context being set up by `SYS_CONTEXT`. Within the duration of a query, the context remains the same for all fetches of the query.

If you (or a function) attempt to change the context within a query, the change will not take effect in the current query. In this way, `SYS_CONTEXT` enables you to store variables in a session.

## Task 2: Create a Unique Context and Associate It with the PL/SQL Package

To perform this task, you use the `CREATE CONTEXT` statement. Each context must have a unique attribute and belong to a namespace. That is, context names must be unique within the database, not just within a schema. Contexts are always owned by the schema `SYS`.

For example:

```
CREATE CONTEXT order_entry USING oe_context;
```

where `order_entry` is the context namespace, and `oe_context` is the trusted package that can set attributes in the context namespace.

After you have created the context, you can set or reset the context attributes by using the `DBMS_SESSION.SET_CONTEXT` package. The values of the attributes you set remain either until you reset them or until the user ends the session.

You can set the context attributes inside only the trusted procedure you named in the `CREATE CONTEXT` statement. This prevents a malicious user from changing context attributes without proper attribute validation.

## Task 3: Set the Context Before the User Retrieves Data

Always use an event trigger on login to pull session information into the context. This sets the user's security-limiting attributes for the database to evaluate, and thus enables it to make the appropriate security decisions.

Other considerations come into play if you have a changing set of books, or if positions change constantly. In these cases, the new attribute values may not be picked up right away, and you must force a cursor reparse to pick it up.

**See Also:** ["Feature Design Principles for Application Context"](#) on page 11-45

["Examples"](#) on page 11-53

## Task 4. Use the Context in a Policy Function

Now that you have set up the context and the PL/SQL package, you can go ahead and have your policy functions use the application context to make policy decisions based on different context values.

## Examples

This section provides three examples, each using application context within a fine-grained access control function.

- [Example 1: Order Entry Application](#)
- [Example 2: Human Resources Application #1](#)
- [Example 3: Human Resources Application #2](#)

### Example 1: Order Entry Application

This simple example uses application context to implement the policy, "Customers can see their own orders only." This example guides you through the following steps in building the application:

- [Step 1. Create a PL/SQL package which sets the context for the application](#)
- [Step 2. Create an application context](#)
- [Step 3. Access the application context inside the package that implements the security policy on the database object](#)
- [Step 4. Create the new security policy](#)

The procedure in this example:

- Assumes a one-to-one relationship between users and customers
- Finds the user's customer number (`Cust_num`)
- Caches the customer number in the application context

You can later refer to the `cust_num` attribute of your order entry context (`order_entry_ctx`) inside the security policy function.

---

---

**Note:** You could use a login trigger to set the initial context.

---

---

**See Also:** Compare and contrast this example, which uses an application context within the dynamically generated predicate, with "[Example of a Dynamically Modified Statement](#)" on page 11-39, which uses a subquery in the predicate.

[Chapter 12, "Using Triggers"](#)

**Step 1. Create a PL/SQL package which sets the context for the application**

```
CREATE OR REPLACE PACKAGE apps.oe_ctx AS
    PROCEDURE set_cust_num ;
END;

CREATE OR REPLACE PACKAGE BODY apps.oe_ctx AS
    PROCEDURE set_cust_num IS
        custnum NUMBER;
    BEGIN
        SELECT cust_no INTO custnum FROM customers WHERE username =
            SYS_CONTEXT('USERENV', 'session_user');
        /* SET cust_num attribute in 'order_entry' context */
        DBMS_SESSION.SET_CONTEXT('order_entry', 'cust_num', custnum);
        DBMS_SESSION.SET_CONTEXT('order_entry', 'cust_num', custnum);
    END set_cust_num;
END;
```

---

---

**Note:** This example does not treat error handling.

You can access predefined attributes—such as session user—by using `SYS_CONTEXT('USERENV', session_primitive)`.

For more information, see *Oracle8i SQL Reference*.

---

---

**Step 2. Create an application context**

Create an application context by entering:

```
CREATE CONTEXT Order_entry USING Apps.Oe_ctx;
```

**Step 3. Access the application context inside the package that implements the security policy on the database object**

---

---

**Note:** You may need to set up the following data structures for certain examples to work:

```
CREATE PACKAGE Oe_security AS
    FUNCTION Custnum_sec (D1 VARCHAR2, D2 VARCHAR2)
    RETURN VARCHAR2;
END;
```

---

---

The package body appends a dynamic predicate to `SELECT` statements on the `ORDERS_TAB` table. This predicate limits the orders returned to those of the user's

customer number by accessing the `cust_num` context attribute, instead of a subquery to the customers table.

```
CREATE OR REPLACE PACKAGE BODY Oe_security AS

/* limits select statements based on customer number: */
FUNCTION Custnum_sec (D1 VARCHAR2, D2 VARCHAR2) RETURN VARCHAR2
IS
    D_predicate VARCHAR2 (2000)
BEGIN
    D_predicate = 'cust_no = SYS_CONTEXT("order_entry", "cust_num")';
    RETURN D_predicate;
END Custnum_sec;
END Oe_security;
```

#### Step 4. Create the new security policy

---



---

**Note:** You may need to set up the following data structures for certain examples to work:

```
CONNECT sys/change_on_install AS sysdba;
CREATE USER securr IDENTIFIED BY securr;
```

---



---

```
DBMS_RLS.ADD_POLICY ('scott', 'orders_tab', 'oe_policy', 'securr',
                    'oe_security.custnum_sec', 'select')
```

This statement adds a policy named `OE_POLICY` to the `ORDERS_TAB` table for viewing in schema `SCOTT`. The `SECUSR.OE_SECURITY.CUSTNUM_SEC` function implements the policy, is stored in the `SECUSR` schema, and applies to `SELECT` statements only.

Now, any select statement by a customer on the `ORDERS_TAB` table automatically returns only that customer's orders. In other words, the dynamic predicate modifies the user's statement from this:

```
SELECT * FROM Orders_tab;
```

to this:

```
SELECT * FROM Orders_tab
    WHERE Custno = SYS_CONTEXT('order_entry', 'cust_num');
```

Note the following with regard to this example:

- In reality, you might have several predicates based on a user's position. For example, a sales rep would be able to see records for all his customers, and an order entry clerk would be able to see any customer order. You could expand the `custnum_sec` function to return different predicates based on the user's position context value.
- The use of application context in a fine-grained access control package effectively gives you a bind variable in a parsed statement. For example:

```
SELECT * FROM Orders_tab
      WHERE Custno = SYS_CONTEXT('order_entry', 'cust_num')
```

This is fully parsed and optimized, but the evaluation of the user's `CUST_NUM` attribute value for the `ORDER_ENTRY` context takes place at execution. This means that you get the benefit of an optimized statement which executes differently for each user executing the statement.

---

---

**Note:** You can improve the performance of the function in this example even more by indexing `CUST_NO`.

---

---

- You could set your context attributes based on data from a database table or tables, or from a directory server using LDAP (Lightweight Directory Access Protocol).

## Example 2: Human Resources Application #1

This example uses application context to control user access by way of a Human Resources application. It guides you through the following three tasks, each of which is described more fully below.

- [Step 1. Create a PL/SQL package with a number of functions that sets the context for the application](#)
- [Step 2. Create the context and associate it with the package](#)
- [Step 3. Create the initialization script for the application](#)

In this example, assume that the application context for the Human Resources application is assigned to the HR\_CTX namespace.

### Step 1. Create a PL/SQL package with a number of functions that sets the context for the application

---



---

**Note:** You may need to set up the following data structures for certain examples to work:

```
CREATE OR REPLACE PACKAGE apps.hr_sec_ctx IS
  PROCEDURE set_resp_id (respid NUMBER);
  PROCEDURE set_org_id (orgid NUMBER);
  /* PROCEDURE validate_resp_id (respid NUMBER); */
  /* PROCEDURE validate_org_id (orgid NUMBER); */
END hr_sec_ctx;
```

---



---

APPS is the schema owning the package.

```
CREATE OR REPLACE PACKAGE BODY apps.hr_sec_ctx IS
  /* function to set responsibility id */
  PROCEDURE set_resp_id (respid NUMBER) IS
  BEGIN

  /* validate respid based on primitive and other context */
  /*   validate_resp_id (respid); */

  /* set resp_id attribute under namespace 'hr_ctx' */
  DEMS_SESSION.SET_CONTEXT('hr_ctx', 'resp_id', respid);
  END set_resp_id;

  /* function to set organization id */
  PROCEDURE set_org_id (orgid NUMBER) IS
```

```
BEGIN
/* validate organization ID */
/*   validate_org_id(orgid); */
/* set org_id attribute under namespace 'hr_ctx' */
   DBMS_SESSION.SET_CONTEXT('hr_ctx', 'org_id', orgid);
END set_org_id;

/* more functions to set other attributes for the HR application */
END hr_sec_ctx;
```

## Step 2. Create the context and associate it with the package

```
CREATE CONTEXT Hr_ctx USING Apps.Hr_sec_ctx;
```

## Step 3. Create the initialization script for the application

Suppose that the execute privilege on the package HR\_SEC\_CTX has been granted to the schema running the application. Part of the script will make calls to set various attributes of the HR\_CTX context. Here, we do not show how the context is determined. Normally, it is based on the primitive context or other derived context.

```
APPS.HR_SEC_CTX.SET_RESP_ID(1);
APPS.HR_SEC_CTX.SET_ORG_ID(101);
```

The SYS\_CONTEXT function can be used for data access control based on this application context. For example, the base table HR\_ORGANIZATION\_UNIT can be secured by a view that restricts access to rows based on attribute ORG\_ID:

---

---

**Caution:** You may need to set up data structures for certain examples to work:

```
CREATE TABLE hr_organization_unit (organization_id NUMBER);
```

---

---

```
CREATE VIEW Hr_organization_secv AS
  SELECT * FROM hr_organization_unit
  WHERE Organization_id = SYS_CONTEXT('hr_ctx', 'org_id');
```

### Example 3: Human Resources Application #2

---



---

**Note:** You may need to set up the following data structures for certain examples to work:

```
CREATE TABLE Payroll(
  Srate NUMBER,
  Orate NUMBER,
  Acctno NUMBER,
  Empno NUMBER,
  Name VARCHAR2(20));
CREATE TABLE Directory_u(
  Empno NUMBER,
  Mgrno NUMBER,
  Rank NUMBER);
CREATE SEQUENCE Empno_seq
CREATE SEQUENCE Rank_seq
```

---



---

This example illustrates the use of the following security features in Oracle8i:

- Event triggers
- Application context
- Fine-grained access control
- Encapsulation of privileges in stored procedures

In this example, we associate a security policy with the table called `DIRECTORY` which has the following columns:

<code>EMPNO</code>	identification number for each employee
<code>MGRID</code>	employee identification number for the manager of each employee
<code>RANK</code>	position of the employee in the corporate hierarchy

The security policy associated with this table has two elements:

- All users can find the `MGRID` for a specific `EMPNO`. To implement this, we create a definer's right package in the Human Resources schema (HR) to do `SELECT` on the table.
- Managers can update the positions in the corporate hierarchy of only their direct subordinates. To do this they must use only the designated application. To implement this:

- \* Define fine-grained access policies on the table based on EMPNO and application context.
- \* Set EMPNO by using a logon trigger.
- \* Set the application context by using the designated package for processing the updates (event triggers and application context).

---

---

**Note:** In this example, we grant UPDATE privileges on the table to public, because fine-grained access control prevents an unauthorized user from wrongly modifying a given row.

---

---

```
CONNECT system/manager AS sysdba
GRANT CONNECT,RESOURCE,UNLIMITED TABLESPACE,CREATE ANY CONTEXT, CREATE
PROCEDURE, CREATE ANY TRIGGER TO HR IDENTIFIED BY HR;
CONNECT hr/hr;
CREATE TABLE Directory (Empno    NUMBER(4) NOT NULL,
                        Mgrno    NUMBER(4) NOT NULL,
                        Rank     NUMBER(7,2) NOT NULL);

CREATE TABLE Payroll (Empno    NUMBER(4) NOT NULL,
                      Name     VARCHAR(30) NOT NULL );

/* seed the tables with a couple of managers: */
INSERT INTO Directory VALUES (1, 1, 1.0);
INSERT INTO Payroll VALUES (1, 'KING');
INSERT INTO Directory VALUES (2, 1, 5);
INSERT INTO Payroll VALUES (2, 'CLARK');

/* Create the sequence number for EMPNO: */
CREATE SEQUENCE Empno_seq START WITH 5;

/* Create the sequence number for RANK: */
CREATE SEQUENCE Rank_seq START WITH 100;

CREATE OR REPLACE CONTEXT Hr_app USING Hr.Hr0_pck;
CREATE OR REPLACE CONTEXT Hr_sec USING Hr.Hr1_pck;

CREATE or REPLACE PACKAGE Hr0_pck IS
PROCEDURE adjustrankby1(Empno NUMBER);
END;

CREATE or REPLACE PACKAGE BODY Hr0_pck IS
/* raise the rank of the empno by 1: */
```

```

PROCEDURE Adjustrankby1(Empno NUMBER)
IS
    Stmt  VARCHAR2(100);
    BEGIN

        /*Set context to indicate application state */
        DBMS_SESSION.SET_CONTEXT('hr_app','adjstate',1);
        /* Now we can issue DML statement: */
        Stmt := 'UPDATE SET Rank := Rank +1 FROM Directory d WHERE d.Empno = '
        || Empno;
        EXECUTE IMMEDIATE STMT;

    /* Re-set application state: */
        DBMS_SESSION.SET_CONTEXT('hr_app','adjstate',0);
        END;
END;

CREATE or REPLACE PACKAGE hr1_pck IS PROCEDURE setid;
END;
/
/* Based on userid, find EMPNO, and set it in application context */

CREATE or REPLACE PACKAGE BODY Hr1_pck IS
PROCEDURE setid
IS
    id NUMBER;
    BEGIN
        SELECT Empno INTO id FROM Payroll WHERE Name =
            SYS_CONTEXT('userenv','session_user') ;
        DBMS_SESSION.SET_CONTEXT('hr_sec','empno',id);
        DBMS_SESSION.SET_CONTEXT('hr_sec','appid',id);
    EXCEPTION
        /* For purposes of demonstration insert into payroll table
        / so that user can continue on and run example. */
        WHEN NO_DATA_FOUND THEN
            INSERT INTO Payroll (Empno, Name)
                VALUES (Empno_seq.NEXTVAL, SYS_CONTEXT('userenv','session_user'));
            INSERT INTO Directory (Empno, Mgrno, Rank)
                VALUES (Empno_seq.CURRVAL, 2, Rank_seq.NEXTVAL);
            SELECT Empno INTO id FROM Payroll WHERE Name =
                sys_context('userenv','session_user') ;
            DBMS_SESSION.SET_CONTEXT('hr_sec','empno',id);
            DBMS_SESSION.SET_CONTEXT('hr_sec','appid',id);
        WHEN OTHERS THEN
            NULL;

```

```
        /* If this is to be fired via a "logon" trigger,
        / you need to handle exceptions if you want the user to continue
        / logging into the database. */
    END;
END;

GRANT EXECUTE ON Hr1_pck TO public;

CONNECT system/manager AS sysdba

CREATE OR REPLACE TRIGGER Databasetrigger

AFTER LOGON
ON DATABASE
BEGIN
    hr.Hr1_pck.Setid;
END;

/* Creates the package for finding the MGRID for a particular EMPNO
using definer's right (encapsulated privileges). Note that users are
granted EXECUTE privileges only on this package, and not on the table
(DIRECTORY) it is querying. */

CREATE or REPLACE PACKAGE hr2_pck IS
    FUNCTION Findmgr(Empno NUMBER) RETURN NUMBER;
END;

CREATE or REPLACE PACKAGE BODY hr2_pck IS
    /* insert a new employee record: */
    FUNCTION findmgr(empno number) RETURN NUMBER IS
        Mgrid NUMBER;
    BEGIN
        SELECT mgrno INTO mgrid FROM directory WHERE mgrid = empno;
    RETURN mgrid;
    END;
END;

CREATE or REPLACE FUNCTION secure_updates(ns varchar2,na varchar2)
RETURN VARCHAR2 IS
    Results VARCHAR2(100);
BEGIN
    /* Only allow updates when designated application has set the session
state to indicate we are inside it. */
    IF (sys_context('hr_sec','adjstate') = 1)
        THEN results := 'mgr = SYS_CONTEXT("hr_sec","empno")';
```

```
        ELSE results := '1=2';  
    END IF;  
    RETURN Results;  
END;
```

```
/* Attaches fine-grained access policy to all update operations on  
hr.directory */
```

```
CONNECT system/manager AS sysdba;  
BEGIN  
    DBMS_RLS.ADD_POLICY('hr','directory_u','secure_update','hr',  
                        'secure_updates','update',TRUE,TRUE);  
END;
```

## Authentication Through a Middle Tier

The growth of three-tier systems (for example, browser to application server to database) has increased dramatically with the growth of the Internet. Indeed, three-tier applications are often referred to as the "Internet computing model."

Oracle8i *n*-tier authentication addresses many security difficulties that arise in three-tier applications. It enables organizations to reap the benefits of Internet computing, while minimizing the security risks of three-tier systems. This section includes:

- [Advantages of \*n\*-Tier Authentication](#)
- [Security Challenges of Three-tier Computing](#)
- [Oracle8i \*n\*-Tier Authentication Solutions](#)

### Advantages of *n*-Tier Authentication

Three-tier systems provide many benefits to organizations.

- Application servers and web servers enable users to access data stored in legacy applications.
- Users like using a familiar, easy-to-use browser interface.
- Organizations can separate application logic from data storage, partitioning the former in application servers and the latter in databases.
- Organizations can also lower their cost of computing by replacing many "fat clients" with a number of "thin clients" and an application server.

In addition, Oracle *n*-tier authentication delivers the following security benefits:

- A limited trust model, by controlling the users on whose behalf middle tiers can connect, and the roles the middle tiers can assume for the user
- Scalability, by supporting lightweight user sessions through OCI, and eliminating the overhead of reauthenticating clients
- Accountability, by preserving the identity of the real user through to the database, and enabling auditing of actions taken on behalf of the real user

---

---

**Note:** Oracle8i supports the above functionality in three tiers only, not across multiple middle tiers.

---

---

## Security Challenges of Three-tier Computing

While three-tier computing provides many benefits, it raises a number of new security issues:

- [Who Is the Real User?](#)
- [Does the Middle Tier Have Too Much Privilege?](#)
- [How to Audit? Whom to Audit?](#)
- [Can the User Be Reauthenticated to the Database?](#)

### Who Is the Real User?

Most organizations want to know the identity of the actual user who is accessing the database, for reasons of access control or auditing. User accountability is diminished if the identity of the users cannot be traced through all tiers of the application.

Furthermore, if only the application server knows who the user is, then all per-user security enforcement must be done by the application itself. Application-based security is very expensive. If each application that accesses the data must enforce security, then security must be reimplemented in each and every application. It is often preferable to build security on the data itself, with per-user accountability enforced within the database.

### **Does the Middle Tier Have Too Much Privilege?**

Some organizations are willing to accept three-tier systems within the enterprise, in which "all-privileged" middle tiers, such as transaction processing (TP) monitors, can perform all actions for all users. In this architecture, the middle tier connects to the database as the same user for all application users. It therefore needs to have *all* privileges that application users need to do their jobs.

This computing model may be undesirable in the Internet, where the middle tier resides outside, on, or just inside a firewall. More desirable, in this context, is a *limited trust model*, in which the identity of the real client is known to the data server, and the application server (or other middle tier) has a restricted privilege set.

Also useful is the ability to limit the users on whose behalf a middle tier can connect, and the roles the middle tier can assume for the user. For example, many organizations would prefer that users have different privileges depending on where they are connecting from. A user connecting to a web server or application server on the firewall might only be able to use very minimal privileges to access data, whereas a user connecting to a web server or application server within the enterprise might be able to exercise all privileges she is otherwise entitled to have.

### **How to Audit? Whom to Audit?**

Accountability through auditing is a basic principle of information security. Most organizations want to know on whose behalf a transaction was accomplished, not just that a particular application server performed a transaction. A system must therefore be able to differentiate between a user performing a transaction, and an application server performing a transaction on behalf of a user.

Auditing in three-tier systems should be tied to the issue of knowing the real user: if you cannot preserve the user's identity through the middle tier of a three-tier application, you cannot audit actions on behalf of the user.

## Can the User Be Reauthenticated to the Database?

In client/server systems, authentication tends to be straightforward: the client authenticates to the server. In three-tier systems authentication is more difficult, because there are several potential authentications.

- [Client to Middle Tier Authentication](#)
- [Middle Tier to Database Authentication](#)
- [Client Reauthentication Through Middle Tier to Database](#)

**Client to Middle Tier Authentication** Client authentication to the middle tier is clearly required if a system is to conform with basic security principles. The middle tier is typically the first gateway to useful information that the user can access. Users *must*, therefore, authenticate to the middle tier. Note that such authentication may be mutual; that is, the middle tier authenticates to the client just as the client authenticates to the middle tier.

**Middle Tier to Database Authentication** Since the middle tier must typically initiate a connection to a database to retrieve data (whether on its own behalf or on behalf of the user), this connection clearly must be authenticated. In fact, the Oracle8i database does not allow unauthenticated connections. Again, middle tier to database authentication may also be mutual.

**Client Reauthentication Through Middle Tier to Database** Client reauthentication from the middle tier to the database is problematic in three-tier systems. The username may not be the same on the middle tier and the database. In this case, users may need to reenter a username and password, which the middle tier uses to connect on their behalf. Or, more commonly, the middle tier may need to map the username provided, to a database username. This mapping is often done in an LDAP-compliant directory service, such as Oracle Internet Directory.

For the client to reauthenticate himself to the database, the middle tier either needs to ask the user for a password (which it then must be trusted to pass to the database), or the middle tier must retrieve a password for the user and use that to authenticate the user. Both approaches involve security risks, because the middle tier is trusted to handle the user's password properly, and not use it maliciously.

One of the only cases for which reauthentication does not involve trusting the middle tier occurs when a middle tier downloads an applet to a client, and the client connects *directly* to the database via the applet. In this case, the application server is literally just that: it serves the application (applet) to the user, and has no part in further authentication of the user.

Reauthenticating the client to the back-end database is not always beneficial. First, two sets of authentication handshakes per user involves considerable network overhead. Second, you must trust the middle tier to have authenticated the user. (You clearly must trust the middle tier if it retrieves or otherwise is privy to the user's password.) It is therefore not unreasonable for the database to simply accept that the middle tier has performed proper authentication. In other words, the database accepts the identity of the real client without requiring the real client to authenticate herself.

For some authentication protocols, client reauthentication is just not possible. For example, many browsers and application servers support the Secure Sockets Layer (SSL) protocol. Both the Oracle8i database (through Oracle Advanced Security) and Oracle Application Server support the use of SSL for client authentication. However, SSL is a point-to-point protocol, not an end-to-end protocol. It cannot be used to reauthenticate a browser client (through the middle tier) to the database.

The reason for this is that a user cannot *securely* give up his private key to the middle tier in order for the reauthentication of the client to occur. Once the user's private key is compromised, the user's very identity is compromised. In addition, there is no way to "tunnel" through a middle tier so that the authentication of the browser client to the database can occur directly.

In short, organizations deploying three-tier systems require flexibility as regards reauthentication of the client. In some cases, they cannot reauthenticate the client; in other cases, they may choose whether or not to reauthenticate the client.

## Oracle8i *n*-Tier Authentication Solutions

The following sections explain how Oracle8i addresses each of the challenges listed above.

- [Passing Through the Identity of the Real User](#)
- [Limiting the Privilege of the Middle Tier](#)
- [Reauthenticating the Real User](#)
- [Auditing Actions Taken on Behalf of the Real User](#)

### Passing Through the Identity of the Real User

Many organizations want to know who the real user is through all tiers of an application, without sacrificing the benefits of a middle tier. Oracle8i provides the ability to preserve client identity through the Oracle Call Interface (OCI).

OCI enables a middle tier to set up, within a single database connection, a number of "lightweight" user sessions, each of which uniquely identifies a connected user. These lightweight sessions reduce the network overhead of creating separate network connections from the middle tier to the database. The application can switch between these sessions as required to process transactions on behalf of users.

The full authentication sequence from the client to the middle tier to the database occurs as follows:

1. The client authenticates to the middle tier, using whatever form of authentication the middle tier will accept. For example, the client could authenticate to the middle tier using a username/password, or an X.509 certificate by means of SSL.
2. The middle tier authenticates itself to Oracle8i, using whatever form of authentication Oracle8i will accept. This could be a password, or an authentication mechanism supported by Oracle Advanced Security, such as a Kerberos ticket or an X.509 certificate (SSL).
3. The middle tier then creates one or more sessions for users using the Oracle Call Interface. The lightweight session information must include username as a minimum. The middle tier may optionally provide a password for the client, and the roles for the client.
4. Since the database cannot require the middle tier to provide a password for the client, authentication is performed by OCI. To create the session for the client, the middle-tier server calls the `OCISessionBegin` function. Prior to calling `OCISessionBegin`, the `OCIAttrSet` function is called to provide the needed

information about the client to the middle tier server. It is called in turn with the following attributes:

<code>OCI_ATTR_USERNAME</code>	Sets the database user name of the client. This attribute is mandatory.
<code>OCI_ATTR_PASSWORD</code>	If the client has provided a database password to be validated by the database, then the middle tier server passes it along with the username. If this attribute is not provided, then it is assumed that the middle tier server has authenticated the client.
<code>OCI_ATTR_PROXY_CREDENTIALS</code>	This attribute tells the server that the client is connecting through a middle tier server.
<code>OCI_ATTR_INITIAL_CLIENT_ROLES</code>	If the middle tier server wants to activate a set of roles upon connecting as the client, then the list is passed along with this attribute.

5. The database verifies that the middle tier is privileged to create sessions on behalf of the user, using the roles provided. (See "[Limiting the Privilege of the Middle Tier](#)," below).

The `OCISessionBegin` call will fail if the application server is not allowed to proxy on behalf of the client by the administrator, or if the application server is not allowed to activate the specified roles.

### Limiting the Privilege of the Middle Tier

"Least privilege" is the principle that users should have the fewest privileges necessary to perform their duties, and no more. As applied to middle tier applications, this means that the middle tier should not have more privileges than it needs. Oracle8i enables you to limit the middle tier such that it can connect only on behalf of certain users, using only specific roles.

For example, suppose that user Sarah wants to connect to the database through a middle tier, `appsrv` (which is also a database user). Sarah has multiple roles, but it is desirable to restrict the middle tier to exercise only the `clerk` role on her behalf.

A DBA could effectively grant permission for `appsrv` to initiate connections on behalf of Sarah using her `clerk` role only, using the following syntax:

```
ALTER USER Sarah GRANT CONNECT THROUGH appsrv WITH ROLE clerk;
```

By default, the middle tier cannot create connections for any client. The permission must be granted on a per-user basis.

To allow `appsrv` to use all of the roles granted to the client Sarah, the following statement would be used:

```
ALTER USER sarah GRANT CONNECT THROUGH appsrv WITH ROLE ALL;
```

Each time a middle tier initiates a lightweight (OCI) session for another database user, the database verifies that the middle tier is privileged to connect for that user, using the role specified.

### Reauthenticating the Real User

As described above, it is not always beneficial to reauthenticate users to the database after they have been authenticated by the middle tier. However, if you wish to do this for an added measure of security, you can pass the database the user's password using the `OCI_ATTR_PASSWORD` attribute of the `OCIAttrSet` call.

**See Also:** For more information about security in three-tier architectures, see the *Oracle Call Interface Programmer's Guide*.

### Auditing Actions Taken on Behalf of the Real User

The *n*-tier authentication features of Oracle8i enable you audit actions a middle tier performs on behalf of a user. For example, suppose an application server `hrappserver` creates multiple lightweight sessions for users Ajit and Jane. A DBA could enable auditing for `SELECTS` on the `bonus` table that `hrappserver` initiates for Jane as follows:

```
AUDIT SELECT ON bonuses BY hrappserver ON BEHALF OF Jane;
```

Alternatively, the DBA could enable auditing on behalf of multiple users (in this case, both Jane and Ajit) connecting through a middle tier as follows:

```
AUDIT SELECT ON bonuses BY hrappserver ON BEHALF OF ANY;
```

This auditing option only audits `SELECT` statements being initiated by `hrappserver` on behalf of other users. A DBA can enable separate auditing options to capture `SELECTS` against the `bonus` table from clients connecting directly to the database:

```
AUDIT SELECT ON bonuses;
```

## Data Encryption

For certain applications, you may decide to encrypt data as an additional measure of security.

Most issues of data security can be handled by appropriate authentication and access control, ensuring that only properly identified and authorized users can access data. Data in the database, however, cannot normally be secured against the database administrator's access, since a DBA has all privileges. Likewise, organizations may have concerns about securing sensitive data stored offline, such as backup files stored with a third party.

Information which may be especially sensitive and warrant encryption could include credit card numbers; national identity numbers in countries with strict privacy laws; or trade secrets, such as industrial formulas. Applications for which a user is authenticated to the application, rather than to the database, may also use encryption to protect the application user password or cookie.

### The DBMS\_OBFUSCATION\_TOOLKIT Package

For applications dealing with this highly sensitive data, Oracle provides the `DBMS_OBFUSCATION_TOOLKIT` PL/SQL package to encrypt and decrypt data, including string inputs and raw inputs. The function is limited to selected algorithms, such as the Data Encryption Standard (DES). Developers may not plug in their own encryption algorithms, and the key length is also fixed. The function prohibits making multiple passes of encryption; that is, you cannot nest encryption calls, thereby encrypting an encrypted value. These restrictions are required by U.S. laws governing the export of cryptographic products.

The cryptographic functions provide encrypt and decrypt services only and do not provide any built-in key management services (such as automatic key recovery). Developers using the package must handle key storage programmatically. For example, encryption keys could be stored apart from the database if the intention is to prevent the DBA from seeing encrypted data.

## Development Considerations

In applications using cryptography, encryption must not interfere with other access controls. That is, it must not prevent users from accessing an object they are otherwise privileged to access. Otherwise, their ability to perform their jobs is impaired. For example, a user who has `SELECT` privilege on `EMP` should not be limited by the encryption mechanism from seeing all the data he is otherwise privileged to see. There is little benefit to encrypting part of a table with one key and part of a table with another key if users need to see all encrypted data in the table; it merely adds to the overhead of decrypting data before users can read it.

Encrypting indexed data is not supported.

Prudent security practice dictates periodically changing an encryption key to mitigate the threat of a compromised key. Changing the key requires that the encrypted object (or objects) be decrypted and reencrypted using the new key or keys. This process may be time-consuming and would probably have to be done when the data is not being accessed.

**See Also:** *PL/SQL User's Guide and Reference*



# Part III

---

## The Active Database

Part III contains the following chapters:

- [Chapter 12, "Using Triggers"](#)
- [Chapter 13, "Working With System Events"](#)
- [Chapter 14, "Using Publish-Subscribe"](#)



---

## Using Triggers

Triggers are procedures that are stored in the database and implicitly run, or *fired*, when something happens.

Traditionally, triggers supported the execution of a PL/SQL block when an `INSERT`, `UPDATE`, or `DELETE` occurred on a table or view. With Oracle8i, triggers support system and other data events on `DATABASE` and `SCHEMA`. Oracle also supports the execution of a PL/SQL or Java procedure.

This chapter discusses DML triggers, `INSTEAD OF` triggers, and system triggers (triggers on `DATABASE` and `SCHEMA`). Topics include:

- [Designing Triggers](#)
- [Creating Triggers](#)
- [Compiling Triggers](#)
- [Modifying Triggers](#)
- [Enabling and Disabling Triggers](#)
- [Listing Information About Triggers](#)
- [Examples of Trigger Applications](#)
- [Triggering Event Publication](#)

## Designing Triggers

Use the following guidelines when designing your triggers:

- Use triggers to guarantee that when a specific operation is performed, related actions are performed.
- Do not define triggers that duplicate the functionality already built into Oracle. For example, do not define triggers to enforce data integrity rules that can be easily enforced using declarative integrity constraints.
- Limit the size of triggers. If the logic for your trigger requires much more than 60 lines of PL/SQL code, then it is better to include most of the code in a stored procedure and call the procedure from the trigger.
- Use triggers only for centralized, global operations that should be fired for the triggering statement, regardless of which user or database application issues the statement.
- **Do not create recursive triggers.** For example, creating an `AFTER UPDATE` statement trigger on the `Emp_tab` table that itself issues an `UPDATE` statement on `Emp_tab`, causes the trigger to fire recursively until it has run out of memory.
- Use triggers on `DATABASE` judiciously. They are executed for *every user every time* the event occurs on which the trigger is created.

## Creating Triggers

Triggers are created using the `CREATE TRIGGER` statement. This statement can be used with any interactive tool, such as SQL\*Plus or Enterprise Manager. When using an interactive tool, a single slash (/) on the last line is necessary to activate the `CREATE TRIGGER` statement.

The following statement creates a trigger for the `Emp_tab` table:

```
CREATE OR REPLACE TRIGGER Print_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON Emp_tab
FOR EACH ROW
WHEN (new.Empno > 0)
DECLARE
    sal_diff number;
BEGIN
    sal_diff := :new.sal - :old.sal;
    dbms_output.put('Old salary: ' || :old.sal);
    dbms_output.put(' New salary: ' || :new.sal);
    dbms_output.put_line(' Difference ' || sal_diff);
END;
/
```

If you enter a SQL statement, such as the following:

```
UPDATE Emp_tab SET sal = sal + 500.00 WHERE deptno = 10;
```

Then, the trigger fires once for each row that is updated, and it prints the new and old salaries, and the difference.

The `CREATE` (or `CREATE OR REPLACE`) statement fails if any errors exist in the PL/SQL block.

---



---

**Note:** The size of the trigger cannot be more than 32K.

---



---

The following sections use this example to illustrate the way that parts of a trigger are specified.

**See Also:** For more realistic examples of `CREATE TRIGGER` statements, see "[Examples of Trigger Applications](#)" on page 12-34.

## Prerequisites for Creating Triggers

Before creating any triggers, run the `CATPROC.SQL` script while connected as `SYS`. This script automatically runs all of the scripts required for, or used within, the procedural extensions to the Oracle Server.

**See Also:** The location of this file is operating system dependent; see your platform-specific Oracle documentation.

## Types of Triggers

A trigger is either a stored PL/SQL block or a PL/SQL, C, or Java procedure associated with a table, view, schema, or the database itself. Oracle automatically executes a trigger when a specified event takes place, which may be in the form of a system event or a DML statement being issued against the table.

Triggers can be:

- DML triggers on tables.
- `INSTEAD OF` triggers on views.

---

---

**Note:** With Oracle8i release 8.1.5, `INSTEAD OF` triggers are only available with the Enterprise Edition. This may change in future releases.

---

---

- System triggers on `DATABASE` or `SCHEMA`: With `DATABASE`, triggers fire for each event for all users; with `SCHEMA`, triggers fire for each event for that specific user.

**See Also:** *Oracle8i SQL Reference* explains the syntax for creating triggers.

## Overview of System Events

You can create triggers to be fired on any of the following:

- DML statements (`DELETE`, `INSERT`, `UPDATE`)
- DDL statements (`CREATE`, `ALTER`, `DROP`)
- Database operations (`SERVERERROR`, `LOGON`, `LOGOFF`, `STARTUP`, `SHUTDOWN`)

## Getting the Attributes of System Events

You can obtain certain event-specific attributes when the trigger is fired.

**See Also:** For a complete list of the functions you can call to get the event attributes, see [Chapter 13, "Working With System Events"](#).

Creating a trigger on `DATABASE` implies that the triggering event is outside the scope of a user (for example, database `STARTUP` and `SHUTDOWN`), and it applies to all users (for example, a trigger created on `LOGON` event by the DBA).

Creating a trigger on `SCHEMA` implies that the trigger is created in the current user's schema and is fired only for that user.

For each trigger, publication can be specified on DML and system events.

**See Also:** ["Triggering Event Publication"](#) on page 12-54.

## Naming Triggers

Trigger names must be unique with respect to other triggers in the same schema. Trigger names do not need to be unique with respect to other schema objects, such as tables, views, and procedures. For example, a table and a trigger can have the same name (however, to avoid confusion, this is not recommended).

## Triggering Statement

The triggering statement specifies the following:

- The type of SQL statement or the system event, database event, or DDL event that fires the trigger body. The options include `DELETE`, `INSERT`, and `UPDATE`. One, two, or all three of these options can be included in the triggering statement specification.
- The table, view, `DATABASE`, or `SCHEMA` associated with the trigger.

---

---

**Note:** Exactly one table or view can be specified in the triggering statement. If the `INSTEAD OF` option is used, then the triggering statement may only specify a view; conversely, if a view is specified in the triggering statement, then only the `INSTEAD OF` option may be used.

---

---

For example, the `PRINT_SALARY_CHANGES` trigger fires after any `DELETE`, `INSERT`, or `UPDATE` on the `Emp_tab` table. Any of the following statements trigger the `PRINT_SALARY_CHANGES` trigger given in the previous example:

```
DELETE FROM Emp_tab;  
INSERT INTO Emp_tab VALUES ( . . . );  
INSERT INTO Emp_tab SELECT . . . FROM . . . ;  
UPDATE Emp_tab SET . . . ;
```

### INSERT Trigger Behavior

`INSERT` triggers will fire during import and during `SQL*Loader` conventional loads. (For direct loads, triggers are disabled before the load.)

For example, you have three tables: A, B, and C. You also have an `INSERT` trigger on table A which looks from table B and inserts into table C. If you import table A, then table C is also updated.

---

---

**Note:** The `IGNORE` parameter determines whether triggers will be fired during import. If `IGNORE=N` (default), then import does not load an already existing table, so no pre-existing triggers will fire. If the table did not exist, then import creates and loads it before any triggers are defined, so again, they do not fire. If `IGNORE=Y`, then import loads rows into existing tables. Triggers will fire, and indexes will be maintained.

---

---

### Column List for UPDATE

If a triggering statement specifies `UPDATE`, then an optional list of columns can be included in the triggering statement. If you include a column list, then the trigger is fired on an `UPDATE` statement only when one of the specified columns is updated. If you omit a column list, then the trigger is fired when any column of the associated table is updated. A column list cannot be specified for `INSERT` or `DELETE` triggering statements.

The previous example of the `PRINT_SALARY_CHANGES` trigger could include a column list in the triggering statement. For example:

```
. . . BEFORE DELETE OR INSERT OR UPDATE OF ename ON Emp_tab . . .
```

### Usage Notes

- You cannot specify a column list for UPDATE with INSTEAD OF triggers.
- If the column specified in the UPDATE OF clause is an object column, then the trigger is also fired if any of the attributes of the object are modified.
- You cannot specify UPDATE OF clauses on collection columns.

## BEFORE and AFTER Options

The BEFORE or AFTER option in the CREATE TRIGGER statement specifies exactly when to fire the trigger body in relation to the triggering statement that is being run. In a CREATE TRIGGER statement, the BEFORE or AFTER option is specified just before the triggering statement. For example, the PRINT\_SALARY\_CHANGES trigger in the previous example is a BEFORE trigger.

---

---

**Note:** AFTER row triggers are slightly more efficient than BEFORE row triggers. With BEFORE row triggers, affected data blocks must be read (logical read, not physical read) once for the trigger and then again for the triggering statement.

Alternatively, with AFTER row triggers, the data blocks must be read only once for both the triggering statement and the trigger.

---

---

## INSTEAD OF Triggers

The INSTEAD OF option can also be used in triggers. INSTEAD OF triggers provide a transparent way of modifying views that cannot be modified directly through UPDATE, INSERT, and DELETE statements. These triggers are called INSTEAD OF triggers because, unlike other types of triggers, Oracle fires the trigger *instead of* executing the triggering statement. The trigger performs UPDATE, INSERT, or DELETE operations directly on the underlying tables.

You can write normal UPDATE, INSERT, and DELETE statements against the view, and the INSTEAD OF trigger works invisibly in the background to make the right actions take place.

INSTEAD OF triggers can only be activated for each row.

**See Also:** ["FOR EACH ROW Option"](#) on page 12-12

### Usage Notes

- With Oracle8i release 8.1.5, `INSTEAD OF` triggers are only available with the Enterprise Edition. This may also be available in the Standard Edition in future releases.
- The `INSTEAD OF` option can *only* be used for triggers created over views.
- The `BEFORE` and `AFTER` options *cannot* be used for triggers created over views.
- The `CHECK` option for views is not enforced when inserts or updates to the view are done using `INSTEAD OF` triggers. The `INSTEAD OF` trigger body must enforce the check.

### Views That Are Not Modifiable

A view cannot be modified by `UPDATE`, `INSERT`, or `DELETE` statements if the view query contains any of the following constructs:

- Set operators
- Group functions
- `GROUP BY`, `CONNECT BY`, or `START WITH` clauses
- The `DISTINCT` operator
- Joins (a subset of join views are updatable)

If a view contains pseudocolumns or expressions, then you can only update the view with an `UPDATE` statement that does not refer to any of the pseudocolumns or expressions.

## INSTEAD OF Trigger Example

**Note:** You may need to set up the following data structures for this example to work:

```
CREATE TABLE Project_tab (
    Prj_level NUMBER,
    Projno    NUMBER,
    Resp_dept NUMBER);
CREATE TABLE Emp_tab (
    Empno     NUMBER NOT NULL,
    Ename     VARCHAR2(10),
    Job       VARCHAR2(9),
    Mgr       NUMBER(4),
    Hiredate  DATE,
    Sal       NUMBER(7,2),
    Comm      NUMBER(7,2),
    Deptno    NUMBER(2) NOT NULL);

CREATE TABLE Dept_tab (
    Deptno    NUMBER(2) NOT NULL,
    Dname     VARCHAR2(14),
    Loc       VARCHAR2(13),
    Mgr_no    NUMBER,
    Dept_type NUMBER);
```

The following example shows an INSTEAD OF trigger for inserting rows into the MANAGER\_INFO view.

```
CREATE OR REPLACE VIEW manager_info AS
    SELECT e.ename, e.empno, d.dept_type, d.deptno, p.prj_level,
           p.projno
    FROM   Emp_tab e, Dept_tab d, Project_tab p
    WHERE  e.empno = d.mgr_no
    AND    d.deptno = p.resp_dept;

CREATE OR REPLACE TRIGGER manager_info_insert
INSTEAD OF INSERT ON manager_info
REFERENCING NEW AS n          -- new manager information

FOR EACH ROW
DECLARE
    rowcnt number;
```

```

BEGIN
    SELECT COUNT(*) INTO rowcnt FROM Emp_tab WHERE empno = :n.empno;
    IF rowcnt = 0 THEN
        INSERT INTO Emp_tab (empno,ename) VALUES (:n.empno, :n.ename);
    ELSE
        UPDATE Emp_tab SET Emp_tab.ename = :n.ename
            WHERE Emp_tab.empno = :n.empno;
    END IF;
    SELECT COUNT(*) INTO rowcnt FROM Dept_tab WHERE deptno = :n.deptno;
    IF rowcnt = 0 THEN
        INSERT INTO Dept_tab (deptno, dept_type)
            VALUES(:n.deptno, :n.dept_type);
    ELSE
        UPDATE Dept_tab SET Dept_tab.dept_type = :n.dept_type
            WHERE Dept_tab.deptno = :n.deptno;
    END IF;
    SELECT COUNT(*) INTO rowcnt FROM Project_tab
        WHERE Project_tab.projno = :n.projno;
    IF rowcnt = 0 THEN
        INSERT INTO Project_tab (projno, prj_level)
            VALUES(:n.projno, :n.prj_level);
    ELSE
        UPDATE Project_tab SET Project_tab.prj_level = :n.prj_level
            WHERE Project_tab.projno = :n.projno;
    END IF;
END;

```

The actions shown for rows being inserted into the `MANAGER_INFO` view first test to see if appropriate rows already exist in the base tables from which `MANAGER_INFO` is derived. The actions then insert new rows or update existing rows, as appropriate. Similar triggers can specify appropriate actions for `UPDATE` and `DELETE`.

## Object Views and INSTEAD OF Triggers

`INSTEAD OF` triggers provide the means to modify object view instances on the client-side through OCI calls.

**See Also:** *Oracle Call Interface Programmer's Guide*

To modify an object materialized by an object view in the client-side object cache and flush it back to the persistent store, you must specify `INSTEAD OF` triggers, unless the object view is modifiable. If the object is read only, then it is not necessary to define triggers to pin it.

## Triggers on Nested Table View Columns

INSTEAD OF triggers can also be created over nested table view columns. These triggers provide a way of updating elements of the nested table. They fire for each nested table element being modified. The row correlation variables inside the trigger correspond to the nested table element. This type of trigger also provides an additional correlation name for accessing the parent row that contains the nested table being modified.

---



---

**Note:** These triggers:

- Can only be defined over nested table columns in views.
  - Fire only when the nested table elements are modified using the THE() or TABLE() clauses. They do not fire when a DML statement is performed on the view.
- 
- 

For example, consider a department view that contains a nested table of employees.

```
CREATE OR REPLACE VIEW Dept_view AS
SELECT d.Deptno, d.Dept_type, d.Dept_name,
       CAST (MULTISET ( SELECT e.Empno, e.Empname, e.Salary
                       FROM Emp_tab e
                       WHERE e.Deptno = d.Deptno) AS Emp_list_ Emplist
FROM Dept_tab d;
```

The CAST (MULTISET..) operator creates a multi-set of employees for each department. Now, if you want to modify the emplist column, which is the nested table of employees, then you can define an INSTEAD OF trigger over the column to handle the operation.

The following example shows how an insert trigger might be written:

```
CREATE OR REPLACE TRIGGER Dept_emplist_tr
  INSTEAD OF INSERT ON NESTED TABLE Emplist OF Dept_view
  REFERENCING NEW AS Employee
  PARENT AS Department
  FOR EACH ROW
BEGIN
  -- The insert on the nested table is translated to an insert on the base table:
  INSERT INTO Emp_tab VALUES (
    :Employee.Empno, :Employee.Empname, :Employee.Salary, :Department.Deptno);
END;
```

Any INSERT into the nested table fires the trigger, and the Emp\_tab table is filled with the correct values. For example:

```
INSERT INTO TABLE (SELECT d.Emplist FROM Dept_view d WHERE Deptno = 10)
VALUES (1001, 'John Glenn', 10000)
```

The :department.deptno correlation variable in this example would have a value of 10.

## FOR EACH ROW Option

The FOR EACH ROW option determines whether the trigger is a *row* trigger or a *statement* trigger. If you specify FOR EACH ROW, then the trigger fires once for each row of the table that is affected by the triggering statement. The absence of the FOR EACH ROW option indicates that the trigger fires only once for each applicable statement, but not separately for each row affected by the statement.

For example, you define the following trigger:

---

---

**Note:** You may need to set up the following data structures for certain examples to work:

```
CREATE TABLE Emp_log (
    Emp_id    NUMBER,
    Log_date  DATE,
    New_salary NUMBER,
    Action    VARCHAR2(20));
```

---

---

```
CREATE OR REPLACE TRIGGER Log_salary_increase
AFTER UPDATE ON Emp_tab
FOR EACH ROW
WHEN (new.Sal > 1000)
BEGIN
    INSERT INTO Emp_log (Emp_id, Log_date, New_salary, Action)
    VALUES (:new.Empno, SYSDATE, :new.SAL, 'NEW SAL');
END;
```

Then, you enter the following SQL statement:

```
UPDATE Emp_tab SET Sal = Sal + 1000.0
WHERE Deptno = 20;
```

If there are five employees in department 20, then the trigger fires five times when this statement is entered, because five rows are affected.

The following trigger fires only once for each UPDATE of the Emp\_tab table:

```
CREATE OR REPLACE TRIGGER Log_emp_update
AFTER UPDATE ON Emp_tab
BEGIN
    INSERT INTO Emp_log (Log_date, Action)
        VALUES (SYSDATE, 'Emp_tab COMMISSIONS CHANGED');
END;
```

**See Also:** For the order of trigger firing, see *Oracle8i Concepts*.

The statement level triggers are useful for performing validation checks for the entire statement.

## WHEN Clause

Optionally, a trigger restriction can be included in the definition of a row trigger by specifying a Boolean SQL expression in a WHEN clause.

---



---

**Note:** A WHEN clause cannot be included in the definition of a statement trigger.

---



---

If included, then the expression in the WHEN clause is evaluated for each row that the trigger affects.

If the expression evaluates to TRUE for a row, then the trigger body is fired on behalf of that row. However, if the expression evaluates to FALSE or NOT TRUE for a row (unknown, as with nulls), then the trigger body is not fired for that row. The evaluation of the WHEN clause does not have an effect on the execution of the triggering SQL statement (in other words, the triggering statement is not rolled back if the expression in a WHEN clause evaluates to FALSE).

For example, in the PRINT\_SALARY\_CHANGES trigger, the trigger body is not run if the new value of Empno is zero, NULL, or negative. In more realistic examples, you might test if one column value is less than another.

The expression in a WHEN clause of a row trigger can include correlation names, which are explained below. The expression in a WHEN clause must be a SQL expression, and it cannot include a subquery. You cannot use a PL/SQL expression (including user-defined functions) in the WHEN clause.

---



---

**Note:** You cannot specify the `WHEN` clause for `INSTEAD OF` triggers.

---



---

## The Trigger Body

The trigger body is a `CALL` procedure or a PL/SQL block that can include SQL and PL/SQL statements. The `CALL` procedure can be either a PL/SQL or a Java procedure that is encapsulated in a PL/SQL wrapper. These statements are run if the triggering statement is entered and if the trigger restriction (if included) evaluates to `TRUE`.

The trigger body for row triggers has some special constructs that can be included in the code of the PL/SQL block: correlation names and the `REFERENCEING` option, and the conditional predicates `INSERTING`, `DELETING`, and `UPDATING`.

---



---

**Note:** The `INSERTING`, `DELETING`, and `UPDATING` conditional predicates cannot be used for the `CALL` procedures; they can only be used in a PL/SQL block.

---



---

**Example 1** This example illustrates how a DBA can monitor all users logging on:

---



---

**Note:** You may need to set up data structures similar to the following for certain examples to work:

```
CONNECT system/manager
GRANT ADMINISTER DATABASE TRIGGER TO scott;
CONNECT scott/tiger
CREATE TABLE audit_table (
    seq number,
    user_at VARCHAR2(10),
    time_now DATE,
    term VARCHAR2(10),
    job VARCHAR2(10),
    proc VARCHAR2(10),
    enum NUMBER);
```

---



---

```

CREATE OR REPLACE PROCEDURE foo (c VARCHAR2) AS
BEGIN
    INSERT INTO Audit_table (user_at) VALUES(c);
END;

CREATE OR REPLACE TRIGGER logontrig AFTER LOGON ON DATABASE
CALL foo (ora_login_user)
/

```

**Example 2** This example illustrates a trigger invoking a Java procedure:

```

CREATE OR REPLACE PROCEDURE Before_delete (Id IN NUMBER, Ename VARCHAR2)
IS language Java
name 'thjvTriggers.beforeDelete (oracle.sql.NUMBER, oracle.sql.CHAR)';

CREATE OR REPLACE TRIGGER Pre_del_trigger BEFORE DELETE ON Tab
FOR EACH ROW
CALL Before_delete (:old.Id, :old.Ename)

```

### thjvTriggers.java

```

import java.sql.*
import java.io.*
import oracle.sql.*
import oracle.oracore.*
public class thjvTriggers
{
    public static void
    beforeDelete (NUMBER old_id, CHAR old_name)
    Throws SQLException, CoreException
    {
        Connection conn = JDBCConnection.defaultConnection();
        Statement stmt = conn.createStatement();
        String sql = "insert into logtab values
        (" + old_id.intValue() + ", '" + old_ename.toString() + "', BEFORE DELETE)";
        stmt.executeUpdate (sql);
        stmt.close();
        return;
    }
}

```

### Accessing Column Values in Row Triggers

Within a trigger body of a row trigger, the PL/SQL code and SQL statements have access to the old and new column values of the current row affected by the

triggering statement. Two correlation names exist for every column of the table being modified: one for the old column value, and one for the new column value. Depending on the type of triggering statement, certain correlation names might not have any meaning.

- A trigger fired by an `INSERT` statement has meaningful access to new column values only. Because the row is being created by the `INSERT`, the old values are null.
- A trigger fired by an `UPDATE` statement has access to both old and new column values for both `BEFORE` and `AFTER` row triggers.
- A trigger fired by a `DELETE` statement has meaningful access to `:old` column values only. Because the row no longer exists after the row is deleted, the `:new` values are `NULL`. However, you cannot modify `:new` values: `ORA-4084` is raised if you try to modify `:new` values.

The new column values are referenced using the `new` qualifier before the column name, while the old column values are referenced using the `old` qualifier before the column name. For example, if the triggering statement is associated with the `Emp_tab` table (with the columns `SAL`, `COMM`, etc.), then you can include statements in the trigger body. For example:

```
IF :new.Sal > 10000 . . .
IF :new.Sal < :old.Sal . . .
```

Old and new values are available in both `BEFORE` and `AFTER` row triggers. A new column value can be assigned in a `BEFORE` row trigger, but not in an `AFTER` row trigger (because the triggering statement takes effect before an `AFTER` row trigger is fired). If a `BEFORE` row trigger changes the value of `new.column`, then an `AFTER` row trigger fired by the same statement sees the change assigned by the `BEFORE` row trigger.

Correlation names can also be used in the Boolean expression of a `WHEN` clause. A colon must precede the `old` and `new` qualifiers when they are used in a trigger's body, but a colon is not allowed when using the qualifiers in the `WHEN` clause or the `REFERENCING` option.

### **INSTEAD OF Triggers on Nested Table View Columns**

In the case of `INSTEAD OF` triggers on nested table view columns, the `new` and `old` qualifiers correspond to the new and old nested table elements. The parent row corresponding to this nested table element can be accessed using the `parent` qualifier. The parent correlation name is meaningful and valid only inside a nested table trigger.

## REFERENCING Option

The `REFERENCING` option can be specified in a trigger body of a row trigger to avoid name conflicts among the correlation names and tables that might be named `old` or `new`. Because this is rare, this option is infrequently used.

For example, assume you have a table named `new` with columns `field1` (number) and `field2` (character). The following `CREATE TRIGGER` example shows a trigger associated with the `new` table that can use correlation names and avoid naming conflicts between the correlation names and the table name:

---



---

**Note:** You may need to set up the following data structures for certain examples to work:

```
CREATE TABLE new (
    field1    NUMBER,
    field2    VARCHAR2(20));
```

---



---

```
CREATE OR REPLACE TRIGGER Print_salary_changes
BEFORE UPDATE ON new
REFERENCING new AS Newest
FOR EACH ROW
BEGIN
    :Newest.Field2 := TO_CHAR (:newest.field1);
END;
```

Notice that the `new` qualifier is renamed to `newest` using the `REFERENCING` option, and it is then used in the trigger body.

## Conditional Predicates

If more than one type of DML operation can fire a trigger (for example, `ON INSERT OR DELETE OR UPDATE OF Emp_tab`), then the trigger body can use the conditional predicates `INSERTING`, `DELETING`, and `UPDATING` to run specific blocks of code, depending on the type of statement that fires the trigger. Assume this is the triggering statement:

```
INSERT OR UPDATE ON Emp_tab
```

Within the code of the trigger body, you can include the following conditions:

```
IF INSERTING THEN . . . END IF;
IF UPDATING THEN . . . END IF;
```

The first condition evaluates to `TRUE` only if the statement that fired the trigger is an `INSERT` statement; the second condition evaluates to `TRUE` only if the statement that fired the trigger is an `UPDATE` statement.

In an `UPDATE` trigger, a column name can be specified with an `UPDATING` conditional predicate to determine if the named column is being updated. For example, assume a trigger is defined as the following:

```
CREATE OR REPLACE TRIGGER . . .
. . . UPDATE OF Sal, Comm ON Emp_tab . . .
BEGIN

. . . IF UPDATING ('SAL') THEN . . . END IF;

END;
```

The code in the `THEN` clause runs only if the triggering `UPDATE` statement updates the `SAL` column. The following statement fires the above trigger and causes the `UPDATING (sal)` conditional predicate to evaluate to `TRUE`:

```
UPDATE Emp_tab SET Sal = Sal + 100;
```

### **Error Conditions and Exceptions in the Trigger Body**

If a predefined or user-defined error condition or exception is raised during the execution of a trigger body, then all effects of the trigger body, as well as the triggering statement, are rolled back (unless the error is trapped by an exception handler). Therefore, a trigger body can prevent the execution of the triggering statement by raising an exception. User-defined exceptions are commonly used in triggers that enforce complex security authorizations or integrity constraints.

The only exception to this is when the event under consideration is database `STARTUP`, `SHUTDOWN`, or `LOGIN` when the user logging in is `SYSTEM`. In these scenarios, only the trigger action is rolled back.

## **Triggers and Handling Remote Exceptions**

A trigger that accesses a remote site cannot do remote exception handling if the network link is unavailable. For example:

```

CREATE OR REPLACE TRIGGER Example
AFTER INSERT ON Emp_tab
FOR EACH ROW
BEGIN
    INSERT INTO Emp_tab@Remote      -- <- compilation fails here
    VALUES ('x');                 --    when dblink is inaccessible
EXCEPTION
    WHEN OTHERS THEN
        INSERT INTO Emp_log
        VALUES ('x');
END;

```

A trigger is compiled when it is created. Thus, if a remote site is unavailable when the trigger must compile, then Oracle cannot validate the statement accessing the remote database, and the compilation fails. The previous example exception statement cannot run, because the trigger does not complete compilation.

Because stored procedures are stored in a compiled form, the work-around for the above example is as follows:

```

CREATE OR REPLACE TRIGGER Example
AFTER INSERT ON Emp_tab
FOR EACH ROW
BEGIN
    Insert_row_proc;
END;

CREATE OR REPLACE PROCEDURE Insert_row_proc AS
BEGIN
    INSERT INTO Emp_tab@Remote
    VALUES ('x');
EXCEPTION
    WHEN OTHERS THEN
        INSERT INTO Emp_log
        VALUES ('x');
END;

```

The trigger in this example compiles successfully and calls the stored procedure, which already has a validated statement for accessing the remote database; thus, when the remote INSERT statement fails because the link is down, the exception is caught.

## Restrictions on Creating Triggers

Coding triggers requires some restrictions that are not required for standard PL/SQL blocks. The following sections discuss these restrictions.

**Trigger Size** The size of a trigger cannot be more than 32K.

**Valid SQL Statements in Trigger Bodies** The body of a trigger can contain DML SQL statements. It can also contain `SELECT` statements, but they must be `SELECT... INTO...` statements or the `SELECT` statement in the definition of a cursor.

DDL statements are not allowed in the body of a trigger. Also, no transaction control statements are allowed in a trigger. `ROLLBACK`, `COMMIT`, and `SAVEPOINT` cannot be used. For system triggers, `{CREATE/ALTER/DROP} TABLE` statements and `ALTER...COMPILE` are allowed.

---

---

**Note:** A procedure called by a trigger cannot run the above transaction control statements, because the procedure runs within the context of the trigger body.

---

---

Statements inside a trigger can reference remote schema objects. However, pay special attention when calling remote procedures from within a local trigger. If a timestamp or signature mismatch is found during execution of the trigger, then the remote procedure is not run, and the trigger is invalidated.

**LONG, LONG RAW, and LOB Datatypes** `LONG`, `LONG RAW`, and `LOB` datatypes in triggers are subject to the following restrictions:

- A SQL statement within a trigger can insert data into a column of `LONG` or `LONG RAW` datatype.
- If data from a `LONG` or `LONG RAW` column can be converted to a constrained datatype (such as `CHAR` and `VARCHAR2`), then a `LONG` or `LONG RAW` column can be referenced in a SQL statement within a trigger. The maximum length for these datatypes is 32000 bytes.
- Variables cannot be declared using the `LONG` or `LONG RAW` datatypes.
- `:NEW` and `:PARENT` cannot be used with `LONG` or `LONG RAW` columns.
- `LOB` values for `:NEW` variables cannot be modified in the trigger body. For example:

```
:NEW.Column := ...
```

---

This is not allowed if `column` is of `LOB` datatype.

---

---

**Note:** Previously, `column`, in this example, would not have been allowed if it was an object, a `varray`, or a nested table. This restriction has been lifted in release 8.1.5.

---

---

**References to Package Variables** If an `UPDATE` or `DELETE` statement detects a conflict with a concurrent `UPDATE`, then Oracle performs a transparent `ROLLBACK` to `SAVEPOINT` and restarts the update. This can occur many times before the statement completes successfully. Each time the statement is restarted, the `BEFORE` statement trigger is fired again. The rollback to savepoint does not undo changes to any package variables referenced in the trigger. The package should include a counter variable to detect this situation.

**Row Evaluation Order** A relational database does not guarantee the order of rows processed by a SQL statement. Therefore, do not create triggers that depend on the order in which rows are processed. For example, do not assign a value to a global package variable in a row trigger if the current value of the global variable is dependent on the row being processed by the row trigger. Also, if global package variables are updated within a trigger, then it is best to initialize those variables in a `BEFORE` statement trigger.

When a statement in a trigger body causes another trigger to be fired, the triggers are said to be *cascading*. Oracle allows up to 32 triggers to cascade at any one time. However, you can effectively limit the number of trigger cascades using the initialization parameter `OPEN_CURSORS`, because a cursor must be opened for every execution of a trigger.

**Trigger Evaluation Order** Although any trigger can run a sequence of operations either in-line or by calling procedures, using multiple triggers of the same type enhances database administration by permitting the modular installation of applications that have triggers on the same tables.

Oracle executes all triggers of the same type before executing triggers of a different type. If you have multiple triggers of the same type on a single table, then Oracle chooses an arbitrary order to execute these triggers.

**See Also:** *Oracle8i Concepts* has more information on the firing order of triggers.

Each subsequent trigger sees the changes made by the previously fired triggers. Each trigger can see the old and new values. The old values are the original values, and the new values are the current values, as set by the most recently fired `UPDATE` or `INSERT` trigger.

To ensure that multiple triggered actions occur in a specific order, you must consolidate these actions into a single trigger (for example, by having the trigger call a series of procedures).

You cannot open a database that contains multiple triggers of the same type if you are using any version of Oracle before release 7.1. You also cannot open such a database if your `COMPATIBLE` initialization parameter is set to a version earlier than 7.1.0. For system triggers, compatibility must be 8.1.0.

**Mutating Tables** A *mutating* table is a table that is currently being modified by an `UPDATE`, `DELETE`, or `INSERT` statement, or it is a table that might need to be updated by the effects of a declarative `DELETE CASCADE` referential integrity constraint. The restrictions on such a table apply only to the session that issued the statement in progress.

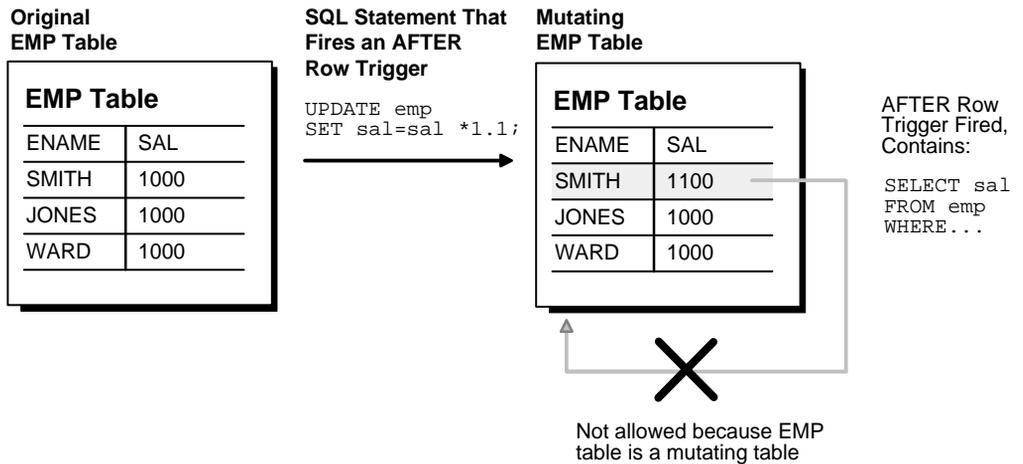
Tables are never considered mutating *for statement triggers* unless the trigger is fired as the result of a `DELETE CASCADE`. Views are not considered mutating in `INSTEAD OF` triggers.

For all row triggers, or for statement triggers that were fired as the result of a `DELETE CASCADE`, there are two important restrictions regarding mutating tables. These restrictions prevent a trigger from seeing an inconsistent set of data.

- The SQL statements of a trigger cannot read from (query) or modify a mutating table of the triggering statement.

Figure 12-1 illustrates the restriction placed on mutating tables.

Figure 12–1 Mutating Tables



Notice that the SQL statement is run for the first row of the table, and then an AFTER row trigger is fired. In turn, a statement in the AFTER row trigger body attempts to query the original table. However, because the EMP table is mutating, this query is not allowed by Oracle. If attempted, then a runtime error occurs, the effects of the trigger body and triggering statement are rolled back, and control is returned to the user or application.

Consider the following trigger:

```
CREATE OR REPLACE TRIGGER Emp_count
AFTER DELETE ON Emp_tab
FOR EACH ROW
DECLARE
    n INTEGER;
BEGIN
    SELECT COUNT(*) INTO n FROM Emp_tab;
    DBMS_OUTPUT.PUT_LINE(' There are now ' || n ||
        ' employees. ');
END;
```

If the following SQL statement is entered:

```
DELETE FROM Emp_tab WHERE Empno = 7499;
```

Then, the following error is returned:

```
ORA-04091: table SCOTT.Emp_tab is mutating, trigger/function may not see it
```

Oracle returns this error when the trigger fires, because the table is mutating when the first row is deleted. (Only one row is deleted by the statement, because Empno is a primary key, but Oracle has no way of knowing that.)

If you delete the line "FOR EACH ROW" from the trigger above, then the trigger becomes a statement trigger, the table is not mutating when the trigger fires, and the trigger does output the correct data.

If you need to update a mutating table, then you could use a temporary table, a PL/SQL table, or a package variable to bypass these restrictions. For example, in place of a single AFTER row trigger that updates the original table, resulting in a mutating table error, you may be able to use two triggers—an AFTER row trigger that updates a temporary table, and an AFTER statement trigger that updates the original table with the values from the temporary table.

Declarative integrity constraints are checked at various times with respect to row triggers.

**See Also:** *Oracle8i Concepts* has information about the interaction of triggers and integrity constraints.

Because declarative referential integrity constraints are currently not supported between tables on different nodes of a distributed database, the mutating table restrictions do not apply to triggers that access remote nodes. These restrictions are also not enforced among tables in the same database that are connected by loop-back database links. A loop-back database link makes a local table appear remote by defining a Net8 path back to the database that contains the link.

You should not use loop-back database links to circumvent the trigger restrictions. Such applications might behave unpredictably.

**Restrictions on Mutating Tables Relaxed** Before Oracle8i, there was a "constraining error" that prevented a row trigger from modifying a table when the parent statement implicitly read that table to enforce a foreign key constraint. As of Oracle8i, there is no constraining error. In addition, checking of the foreign key is deferred until at least the end of the parent statement. The mutating error still prevents the trigger from reading or modifying the table that the parent statement is modifying.

This allows most foreign key constraint actions to be implemented via their obvious after-row trigger, providing the constraint is not self-referential. Update cascade,

update set null, update set default, delete set default, inserting a missing parent, and maintaining a count of children can all be implemented easily. For example, this is an implementation of update cascade:

```
create table p (p1 number constraint ppk primary key);
create table f (f1 number constraint ffk references p);
create trigger pt after update on p for each row begin
    update f set f1 = :new.p1 where f1 = :old.p1;
end;
/
```

This implementation requires care for multirow updates. For example, if a table `p` has three rows with the values (1), (2), (3), and table `f` also has three rows with the values (1), (2), (3), then the following statement updates `p` correctly but causes problems when the trigger updates `f`:

```
update p set p1 = p1+1;
```

The statement first updates (1) to (2) in `p`, and the trigger updates (1) to (2) in `f`, leaving two rows of value (2) in `f`. Then the statement updates (2) to (3) in `p`, and the trigger updates both rows of value (2) to (3) in `f`. Finally, the statement updates (3) to (4) in `p`, and the trigger updates all three rows in `f` from (3) to (4). The relationship of the data in `p` and `f` is lost.

To avoid this problem, you must forbid multirow updates to `p` that change the primary key and reuse existing primary key values. It could also be solved by tracking which foreign key values have already been updated, then modifying the trigger so that no row is updated twice.

That is the only problem with this technique for foreign key updates. The trigger cannot miss rows that have been changed but not committed by another transaction, because the foreign key constraint guarantees that no matching foreign key rows are locked before the after-row trigger is called.

## System Trigger Restrictions

**Nature of the Event** Depending on the event, the publication functionality imposes different restrictions. It may not be possible for the server to impose all restrictions. The restrictions that cannot be fully enforced are clearly documented. For example, certain DDL operations may not be allowed on DDL events.

Only committed triggers are fired. For example, if you create a trigger that should be fired after all `CREATE` events, then the trigger itself does not fire after the creation, because the correct information about this trigger was not committed at

the time when the trigger on `CREATE` events was fired. On the other hand, if you `DROP` a trigger that should be fired *before* all `DROP` events, then the trigger fires before the `DROP`.

For example, if you execute the following SQL statement:

```
CREATE OR REPLACE TRIGGER Foo AFTER CREATE ON DATABASE
BEGIN null;
END;
```

Then, trigger `foo` is not fired after the creation of `foo`. Oracle does not fire a trigger that is not committed.

**See Also:** For other restrictions, see "[List of Events](#)" on page 13-6.

**Foreign Function Callouts** All restrictions on foreign function callouts will also apply.

## Who Is the Trigger User?

If you enter the following statement:

```
SELECT Username FROM USER_USERS;
```

Then, in a trigger, the name of the owner of the trigger is returned, not the name of user who is updating the table.

## Privileges

### Privileges to Create Triggers

To create a trigger in your schema, you must have the `CREATE TRIGGER` system privilege, and either:

- Own the table specified in the triggering statement, or
- Have the `ALTER` privilege for the table in the triggering statement, or
- Have the `ALTER ANY TABLE` system privilege

To create a trigger in another user's schema, you must have the `CREATE ANY TRIGGER` system privilege. With this privilege, the trigger can be created in any schema and can be associated with any user's table. In addition, the user creating the trigger must also have `EXECUTE` privilege on the referenced procedures, functions, or packages.

To create a trigger on DATABASE, you must have the ADMINISTER DATABASE TRIGGER privilege. If this privilege is later revoked, then you can drop the trigger, but not alter it.

### **Privileges for Referenced Schema Objects**

The object privileges to the schema objects referenced in the trigger body must be granted to the trigger's owner explicitly (not through a role). The statements in the trigger body operate under the privilege domain of the trigger's owner, not the privilege domain of the user issuing the triggering statement. This is similar to stored procedures.

## Compiling Triggers

Triggers are similar to PL/SQL anonymous blocks with the addition of the `:new` and `:old` capabilities, but their compilation is different. A PL/SQL anonymous block is compiled each time it is loaded into memory. Compilation involves three stages:

1. Syntax checking: PL/SQL syntax is checked, and a parse tree is generated.
2. Semantic checking: Type checking and further processing on the parse tree.
3. Code generation: The pcode is generated.

Triggers, in contrast, are fully compiled when the `CREATE TRIGGER` statement is entered, and the pcode is stored in the data dictionary. Hence, firing the trigger no longer requires the opening of a shared cursor to run the trigger action. Instead, the trigger is executed directly.

If errors occur during the compilation of a trigger, then the trigger is still created. If a DML statement fires this trigger, then the DML statement fails. (Runtime that trigger errors always cause the DML statement to fail.) You can use the `SHOW ERRORS` statement in SQL\*Plus or Enterprise Manager to see any compilation errors when you create a trigger, or you can `SELECT` the errors from the `USER_ERRORS` view.

## Dependencies

Compiled triggers have dependencies. They become invalid if a depended-on object, such as a stored procedure or function called from the trigger body, is modified. Triggers that are invalidated for dependency reasons are recompiled when next invoked.

You can examine the `ALL_DEPENDENCIES` view to see the dependencies for a trigger. For example, the following statement shows the dependencies for the triggers in the `SCOTT` schema:

```
SELECT NAME, REFERENCED_OWNER, REFERENCED_NAME, REFERENCED_TYPE
       FROM ALL_DEPENDENCIES
       WHERE OWNER = 'SCOTT' and TYPE = 'TRIGGER';
```

Triggers may depend on other functions or packages. If the function or package specified in the trigger is dropped, then the trigger is marked invalid. An attempt is made to validate the trigger on occurrence of the event. If the trigger cannot be validated successfully, then it is marked `VALID WITH ERRORS`, and the event fails.

---

---

**Note:**

- There is an exception for `STARTUP` events: `STARTUP` events succeed even if the trigger fails. There are also exceptions for `SHUTDOWN` events and for `LOGON` events if you login as `SYSTEM`.
  - Because the `DBMS_AQ` package is used to enqueue a message, dependency between triggers and queues cannot be maintained.
- 
- 

## Recompiling Triggers

Use the `ALTER TRIGGER` statement to recompile a trigger manually. For example, the following statement recompiles the `PRINT_SALARY_CHANGES` trigger:

```
ALTER TRIGGER Print_salary_changes COMPILE;
```

To recompile a trigger, you must own the trigger or have the `ALTER ANY TRIGGER` system privilege.

## Migration Issues

Non-compiled triggers cannot be fired under compiled trigger releases (such as Oracle 7.3 and Oracle8). If you are upgrading from a non-compiled trigger release to a compiled trigger release, then all existing triggers must be compiled. The upgrade script `cat73xx.sql` invalidates all triggers, so that they are automatically recompiled when first run. (The `xx` stands for a variable minor release number.)

Downgrading from Oracle 7.3 or later to a release prior to 7.3 requires that you run the `cat73xxd.sql` downgrade script. This handles portability issues between stored and non-stored trigger releases.

## Modifying Triggers

Like a stored procedure, a trigger cannot be explicitly altered: It must be replaced with a new definition. (The `ALTER TRIGGER` statement is used only to recompile, enable, or disable a trigger.)

When replacing a trigger, you must include the `OR REPLACE` option in the `CREATE TRIGGER` statement. The `OR REPLACE` option is provided to allow a new version of an existing trigger to replace the older version, without affecting any grants made for the original version of the trigger.

Alternatively, the trigger can be dropped using the `DROP TRIGGER` statement, and you can rerun the `CREATE TRIGGER` statement.

To drop a trigger, the trigger must be in your schema, or you must have the `DROP ANY TRIGGER` system privilege.

## Debugging Triggers

You can debug a trigger using the same facilities available for stored procedures.

**See Also:** ["Debugging Stored Procedures"](#) on page 9-47

## Enabling and Disabling Triggers

A trigger can be in one of two distinct modes:

- |          |   |
|----------|---|
| Enabled  | An enabled trigger executes its trigger body if a triggering statement is entered and the trigger restriction (if any) evaluates to <code>TRUE</code> .               |
| Disabled | A disabled trigger does not execute its trigger body, even if a triggering statement is entered and the trigger restriction (if any) evaluates to <code>TRUE</code> . |

### Enabling Triggers

By default, a trigger is automatically enabled when it is created; however, it can later be disabled. After you have completed the task that required the trigger to be disabled, re-enable the trigger, so that it fires when appropriate.

Enable a disabled trigger using the `ALTER TRIGGER` statement with the `ENABLE` option. To enable the disabled trigger named `REORDER` of the `INVENTORY` table, enter the following statement:

```
ALTER TRIGGER Reorder ENABLE;
```

All triggers defined for a specific table can be enabled with one statement using the `ALTER TABLE` statement with the `ENABLE` clause with the `ALL TRIGGERS` option. For example, to enable all triggers defined for the `INVENTORY` table, enter the following statement:

```
ALTER TABLE Inventory  
    ENABLE ALL TRIGGERS;
```

### Disabling Triggers

You might temporarily disable a trigger if:

- An object it references is not available.
- You need to perform a large data load, and you want it to proceed quickly without firing triggers.
- You are reloading data.

By default, triggers are enabled when first created. Disable a trigger using the `ALTER TRIGGER` statement with the `DISABLE` option.

For example, to disable the trigger named REORDER of the INVENTORY table, enter the following statement:

```
ALTER TRIGGER Reorder DISABLE;
```

All triggers associated with a table can be disabled with one statement using the ALTER TABLE statement with the DISABLE clause and the ALL TRIGGERS option. For example, to disable all triggers defined for the INVENTORY table, enter the following statement:

```
ALTER TABLE Inventory  
  DISABLE ALL TRIGGERS;
```

## Listing Information About Triggers

The following data dictionary views reveal information about triggers:

- USER\_TRIGGERS
- ALL\_TRIGGERS
- DBA\_TRIGGERS

The new column, BASE\_OBJECT\_TYPE, specifies whether the trigger is based on DATABASE, SCHEMA, table, or view. The old column, TABLE\_NAME, is null if the base object is not table or view.

The column ACTION\_TYPE specifies whether the trigger is a call type trigger or a PL/SQL trigger.

The column TRIGGER\_TYPE includes two additional values: BEFORE EVENT and AFTER EVENT, applicable only to system events.

The column TRIGGERING\_EVENT includes all system and DML events.

**See Also:** The *Oracle8i Reference* provides a complete description of these data dictionary views.

For example, assume the following statement was used to create the REORDER trigger:

---

---

**Caution:** You may need to set up data structures for certain examples to work:

---

---

```

CREATE OR REPLACE TRIGGER Reorder
AFTER UPDATE OF Parts_on_hand ON Inventory
FOR EACH ROW
WHEN(new.Parts_on_hand < new.Reorder_point)
DECLARE
    x NUMBER;
BEGIN
    SELECT COUNT(*) INTO x
    FROM Pending_orders
    WHERE Part_no = :new.Part_no;
    IF x = 0 THEN
        INSERT INTO Pending_orders
        VALUES (:new.Part_no, :new.Reorder_quantity,
            sysdate);
    END IF;
END;

```

The following two queries return information about the REORDER trigger:

```

SELECT Trigger_type, Triggering_event, Table_name
FROM USER_TRIGGERS
WHERE Trigger_name = 'REORDER';

```

TYPE	TRIGGERING_STATEMENT	TABLE_NAME
AFTER EACH ROW	UPDATE	INVENTORY

```

SELECT Trigger_body
FROM USER_TRIGGERS
WHERE Trigger_name = 'REORDER';

```

TRIGGER\_BODY

```

-----
DECLARE
    x NUMBER;
BEGIN
    SELECT COUNT(*) INTO x
    FROM Pending_orders
    WHERE Part_no = :new.Part_no;
    IF x = 0
    THEN INSERT INTO Pending_orders
        VALUES (:new.Part_no, :new.Reorder_quantity,
            sysdate);
    END IF;
END;

```

## Examples of Trigger Applications

You can use triggers in a number of ways to customize information management in an Oracle database. For example, triggers are commonly used to:

- Provide sophisticated auditing
- Prevent invalid transactions
- Enforce referential integrity (either those actions not supported by declarative integrity constraints or across nodes in a distributed database)
- Enforce complex business rules
- Enforce complex security authorizations
- Provide transparent event logging
- Automatically generate derived column values
- Enable building complex views that are updatable
- Track system events

This section provides an example of each of the above trigger applications. These examples are not meant to be used exactly as written: They are provided to assist you in designing your own triggers.

### Auditing with Triggers

Triggers are commonly used to supplement the built-in auditing features of Oracle. Although triggers can be written to record information similar to that recorded by the `AUDIT` statement, triggers should be used only when more detailed audit information is required. For example, use triggers to provide value-based auditing on a per-row basis tables.

Sometimes, the Oracle `AUDIT` statement is considered a *security* audit facility, while triggers can provide *financial* audit facility.

When deciding whether to create a trigger to audit database activity, consider what Oracle's auditing features provide, compared to auditing defined by triggers.

DML and DDL Auditing	Standard auditing options permit auditing of DML and DDL statements regarding all types of schema objects and structures. Comparatively, <i>triggers</i> permit auditing of DML statements entered against tables, and DDL auditing at SCHEMA or DATABASE level.
Centralized Audit Trail	All database audit information is recorded centrally and automatically using the auditing features of Oracle.
Declarative Method	Auditing features enabled using the standard Oracle features are easier to declare and maintain, and less prone to errors, when compared to auditing functions defined by triggers.
Auditing Options can be Audited	Any changes to existing auditing options can also be audited to guard against malicious database activity.
Session and Execution time Auditing	Using the database auditing features, records can be generated once every time an audited statement is entered (BY ACCESS) or once for every session that enters an audited statement (BY SESSION). Triggers cannot audit by session; an audit record is generated each time a trigger-audited table is referenced.
Auditing of Unsuccessful Data Access	Database auditing can be set to audit when unsuccessful data access occurs. However, unless autonomous transactions are used, any audit information generated by a trigger is rolled back if the triggering statement is rolled back. For more information on autonomous transactions, see <i>Oracle8i Concepts</i> .
Sessions can be Audited	Connections and disconnections, as well as session activity (physical I/Os, logical I/Os, deadlocks, etc.), can be recorded using standard database auditing.

When using triggers to provide sophisticated auditing, `AFTER` triggers are normally used. By using `AFTER` triggers, auditing information is recorded after the triggering statement is subjected to any applicable integrity constraints, preventing cases where the audit processing is carried out unnecessarily for statements that generate exceptions to integrity constraints.

When to use `AFTER row` vs. `AFTER statement` triggers depends on the information being audited. For example, row triggers provide value-based auditing on a per-row basis for tables. Triggers can also require the user to supply a "reason code" for issuing the audited SQL statement, which can be useful in both row and statement-level auditing situations.

The following example demonstrates a trigger that audits modifications to the `Emp_tab` table on a per-row basis. It requires that a "reason code" be stored in a global package variable before the update. This shows how triggers can be used to provide value-based auditing and how to use public package variables.

---

---

**Note:** You may need to set up the following data structures for the examples to work:

```
CREATE OR REPLACE PACKAGE Auditpackage AS
    Reason VARCHAR2(10);
PROCEDURE Set_reason(Reason VARCHAR2);
END;
CREATE TABLE Emp99 (
    Empno          NOT NULL    NUMBER(4)
    Ename          VARCHAR2(10)
    Job            VARCHAR2(9)
    Mgr            NUMBER(4)
    Hiredate       DATE
    Sal            NUMBER(7,2)
    Comm           NUMBER(7,2)
    Deptno         NUMBER(2)
    Bonus         NUMBER
    Ssn            NUMBER
    Job_classification NUMBER);

CREATE TABLE Audit_employee (
    Oldssn         NUMBER
    Oldname        VARCHAR2(10)
    Oldjob         VARCHAR2(2)
    Oldsal         NUMBER
    Newssn         NUMBER
    Newname        VARCHAR2(10)
    Newjob        VARCHAR2(2)
    Newsal        NUMBER
    Reason         VARCHAR2(10)
    User1          VARCHAR2(10)
    Systemdate     DATE);
```

---

---

```

CREATE OR REPLACE TRIGGER Audit_employee
AFTER INSERT OR DELETE OR UPDATE ON Emp99
FOR EACH ROW
BEGIN
  /* AUDITPACKAGE is a package with a public package
  variable REASON. REASON could be set by the
  application by a command such as EXECUTE
  AUDITPACKAGE.SET_REASON(reason_string). Note that a
  package variable has state for the duration of a
  session and that each session has a separate copy of
  all package variables. */

  IF Auditpackage.Reason IS NULL THEN
    Raise_application_error(-20201, 'Must specify reason'
      || ' with AUDITPACKAGE.SET_REASON(Reason_string)');
  END IF;

  /* If the above conditional evaluates to TRUE, the
  user-specified error number and message is raised,
  the trigger stops execution, and the effects of the
  triggering statement are rolled back. Otherwise, a
  new row is inserted into the predefined auditing
  table named AUDIT_EMPLOYEE containing the existing
  and new values of the Emp_tab table and the reason code
  defined by the REASON variable of AUDITPACKAGE. Note
  that the "old" values are NULL if triggering
  statement is an INSERT and the "new" values are NULL
  if the triggering statement is a DELETE. */

  INSERT INTO Audit_employee VALUES
    (:old.Ssn, :old.Ename, :old.Job_classification, :old.Sal,
    :new.Ssn, :new.Ename, :new.Job_classification, :new.Sal,
    auditpackage.Reason, User, Sysdate );
END;

```

Optionally, you can also set the reason code back to NULL if you wanted to force the reason code to be set for every update. The following simple AFTER statement trigger sets the reason code back to NULL after the triggering statement is run:

```

CREATE OR REPLACE TRIGGER Audit_employee_reset
AFTER INSERT OR DELETE OR UPDATE ON Emp_tab
BEGIN
  auditpackage.set_reason(NULL);
END;

```

Notice that the previous two triggers are both fired by the same type of SQL statement. However, the AFTER row trigger is fired once for each row of the table affected by the triggering statement, while the AFTER statement trigger is fired only once after the triggering statement execution is completed.

Another example of using triggers to do auditing is shown below. This trigger tracks changes made to the Emp\_tab table and stores this information in AUDIT\_TABLE and AUDIT\_TABLE\_VALUES.

---

---

**Note:** You may need to set up the following data structures for the example to work:

```
CREATE TABLE Audit_table (  
    Seq      NUMBER,  
    User_at  VARCHAR2(10),  
    Time_now DATE,  
    Term     VARCHAR2(10),  
    Job      VARCHAR2(10),  
    Proc     VARCHAR2(10),  
    enum     NUMBER);  
CREATE SEQUENCE Audit_seq;  
CREATE TABLE Audit_table_values (  
    Seq      NUMBER,  
    Dept     NUMBER,  
    Dept1    NUMBER,  
    Dept2    NUMBER);
```

---

---

```

CREATE OR REPLACE TRIGGER Audit_emp
AFTER INSERT OR UPDATE OR DELETE ON Emp_tab
FOR EACH ROW
DECLARE
    Time_now DATE;
    Terminal CHAR(10);
BEGIN
    -- get current time, and the terminal of the user:
    Time_now := SYSDATE;
    Terminal := USERENV('TERMINAL');
    -- record new employee primary key
    IF INSERTING THEN
        INSERT INTO Audit_table
            VALUES (Audit_seq.NEXTVAL, User, Time_now,
                Terminal, 'Emp_tab', 'INSERT', :new.Empno);
    -- record primary key of the deleted row:
    ELSIF DELETING THEN
        INSERT INTO Audit_table
            VALUES (Audit_seq.NEXTVAL, User, Time_now,
                Terminal, 'Emp_tab', 'DELETE', :old.Empno);
    -- for updates, record the primary key
    -- of the row being updated:
    ELSE
        INSERT INTO Audit_table
            VALUES (audit_seq.NEXTVAL, User, Time_now,
                Terminal, 'Emp_tab', 'UPDATE', :old.Empno);
    -- and for SAL and DEPTNO, record old and new values:
    IF UPDATING ('SAL') THEN
        INSERT INTO Audit_table_values
            VALUES (Audit_seq.CURRVAL, 'SAL',
                :old.Sal, :new.Sal);

        ELSIF UPDATING ('DEPTNO') THEN
            INSERT INTO Audit_table_values
                VALUES (Audit_seq.CURRVAL, 'DEPTNO',
                    :old.Deptno, :new.DEPTNO);
        END IF;
    END IF;
END;

```

## Integrity Constraints and Triggers

Triggers and declarative integrity constraints can both be used to constrain data input. However, triggers and integrity constraints have significant differences.

Declarative integrity constraints are statements about the database that are always true. A constraint applies to existing data in the table and any statement that manipulates the table.

**See Also:** [Chapter 4, "Maintaining Data Integrity"](#)

Triggers constrain what a transaction can do. A trigger does not apply to data loaded before the definition of the trigger; therefore, it is not known if all data in a table conforms to the rules established by an associated trigger.

Although triggers can be written to enforce many of the same rules supported by Oracle's declarative integrity constraint features, triggers should only be used to enforce complex business rules that cannot be defined using standard integrity constraints. The declarative integrity constraint features provided with Oracle offer the following advantages when compared to constraints defined by triggers:

Centralized Integrity Checks	All points of data access must adhere to the global set of rules defined by the integrity constraints corresponding to each schema object.
Declarative Method	Constraints defined using the standard integrity constraint features are much easier to write and are less prone to errors, when compared with comparable constraints defined by triggers.

While most aspects of data integrity can be defined and enforced using declarative integrity constraints, triggers can be used to enforce complex business constraints not definable using declarative integrity constraints. For example, triggers can be used to enforce:

- UPDATE and DELETE SET NULL, and UPDATE and DELETE SET DEFAULT referential actions.
- Referential integrity when the parent and child tables are on different nodes of a distributed database.
- Complex check constraints not definable using the expressions allowed in a CHECK constraint.

## Referential Integrity Using Triggers

Many cases of referential integrity can be enforced using triggers. However, only use triggers when you want to enforce the `UPDATE` and `DELETE SET NULL` (when referenced data is updated or deleted, all associated dependent data is set to `NULL`), and `UPDATE` and `DELETE SET DEFAULT` (when referenced data is updated or deleted, all associated dependent data is set to a default value) referential actions, or when you want to enforce referential integrity between parent and child tables on different nodes of a distributed database.

When using triggers to maintain referential integrity, declare the `PRIMARY` (or `UNIQUE`) `KEY` constraint in the parent table. If referential integrity is being maintained between a parent and child table in the same database, then you can also declare the foreign key in the child table, but disable it; this prevents the corresponding `PRIMARY KEY` constraint from being dropped (unless the `PRIMARY KEY` constraint is explicitly dropped with the `CASCADE` option).

To maintain referential integrity using triggers:

- A trigger must be defined for the child table that guarantees values inserted or updated in the foreign key correspond to values in the parent key.
- One or more triggers must be defined for the parent table. These triggers guarantee the desired referential action (`RESTRICT`, `CASCADE`, or `SET NULL`) for values in the foreign key when values are updated or deleted in the parent key. No action is required for inserts into the parent table (no dependent foreign keys exist).

The following sections provide examples of the triggers necessary to enforce referential integrity. The `Emp_tab` and `Dept_tab` table relationship is used in these examples.

Several of the triggers include statements that lock rows (`SELECT... FOR UPDATE`). This operation is necessary to maintain concurrency as the rows are being processed.

**Foreign Key Trigger for Child Table** The following trigger guarantees that before an `INSERT` or `UPDATE` statement affects a foreign key value, the corresponding value exists in the parent key. The mutating table exception included in the example below allows this trigger to be used with the `UPDATE SET DEFAULT` and `UPDATE CASCADE` triggers. This exception can be removed if this trigger is used alone.

```
CREATE OR REPLACE TRIGGER Emp_dept_check
BEFORE INSERT OR UPDATE OF Deptno ON Emp_tab
FOR EACH ROW WHEN (new.Deptno IS NOT NULL)

-- Before a row is inserted, or DEPTNO is updated in the Emp_tab
-- table, fire this trigger to verify that the new foreign
-- key value (DEPTNO) is present in the Dept_tab table.
DECLARE
    Dummy          INTEGER; -- used for cursor fetch below
    Invalid_department EXCEPTION;
    Valid_department EXCEPTION;
    Mutating_table EXCEPTION;
    PRAGMA EXCEPTION_INIT (Mutating_table, -4091);

-- Cursor used to verify parent key value exists.  If
-- present, lock parent key's row so it can't be
-- deleted by another transaction until this
-- transaction is committed or rolled back.
    CURSOR Dummy_cursor (Dn NUMBER) IS
        SELECT Deptno FROM Dept_tab
           WHERE Deptno = Dn
           FOR UPDATE OF Deptno;
BEGIN
    OPEN Dummy_cursor (:new.Deptno);
    FETCH Dummy_cursor INTO Dummy;

-- Verify parent key.  If not found, raise user-specified
-- error number and message.  If found, close cursor
-- before allowing triggering statement to complete:
    IF Dummy_cursor%NOTFOUND THEN
        RAISE Invalid_department;
    ELSE
        RAISE valid_department;
    END IF;
    CLOSE Dummy_cursor;
EXCEPTION
    WHEN Invalid_department THEN
        CLOSE Dummy_cursor;
        Raise_application_error(-20000, 'Invalid Department'
            || ' Number' || TO_CHAR(:new.deptno));
    WHEN Valid_department THEN
        CLOSE Dummy_cursor;
    WHEN Mutating_table THEN
        NULL;
END;
```

**UPDATE and DELETE RESTRICT Trigger for Parent Table** The following trigger is defined on the DEPT\_TAB table to enforce the UPDATE and DELETE RESTRICT referential action on the primary key of the DEPT\_TAB table:

```

CREATE OR REPLACE TRIGGER Dept_restrict
BEFORE DELETE OR UPDATE OF Deptno ON Dept_tab
FOR EACH ROW

-- Before a row is deleted from Dept_tab or the primary key
-- (DEPTNO) of Dept_tab is updated, check for dependent
-- foreign key values in Emp_tab; rollback if any are found.
DECLARE
    Dummy                INTEGER;          -- used for cursor fetch below
    Employees_present     EXCEPTION;
    employees_not_present EXCEPTION;

    -- Cursor used to check for dependent foreign key values.
    CURSOR Dummy_cursor (Dn NUMBER) IS
        SELECT Deptno FROM Emp_tab WHERE Deptno = Dn;

BEGIN
    OPEN Dummy_cursor (:old.Deptno);
    FETCH Dummy_cursor INTO Dummy;
    -- If dependent foreign key is found, raise user-specified
    -- error number and message. If not found, close cursor
    -- before allowing triggering statement to complete.
    IF Dummy_cursor%FOUND THEN
        RAISE Employees_present;          -- dependent rows exist
    ELSE
        RAISE employees_not_present;     -- no dependent rows
    END IF;
    CLOSE Dummy_cursor;

EXCEPTION
    WHEN Employees_present THEN
        CLOSE Dummy_cursor;
        Raise_application_error(-20001, 'Employees Present in'
            || ' Department ' || TO_CHAR(:old.DEPTNO));
    WHEN Employees_not_present THEN
        CLOSE Dummy_cursor;

END;
```

---

---

**Caution: This trigger does not work with self-referential tables (tables with both the primary/unique key and the foreign key). Also, this trigger does not allow triggers to cycle (such as, A fires B fires A).**

---

---

**UPDATE and DELETE SET NULL Triggers for Parent Table** The following trigger is defined on the DEPT\_TAB table to enforce the UPDATE and DELETE SET NULL referential action on the primary key of the DEPT\_TAB table:

```
CREATE OR REPLACE TRIGGER Dept_set_null
AFTER DELETE OR UPDATE OF Deptno ON Dept_tab
FOR EACH ROW

-- Before a row is deleted from Dept_tab or the primary key
-- (DEPTNO) of Dept_tab is updated, set all corresponding
-- dependent foreign key values in Emp_tab to NULL:
BEGIN
    IF UPDATING AND :OLD.Deptno != :NEW.Deptno OR DELETING THEN
        UPDATE Emp_tab SET Emp_tab.Deptno = NULL
            WHERE Emp_tab.Deptno = :old.Deptno;
    END IF;
END;
```

**DELETE Cascade Trigger for Parent Table** The following trigger on the DEPT\_TAB table enforces the DELETE CASCADE referential action on the primary key of the DEPT\_TAB table:

```
CREATE OR REPLACE TRIGGER Dept_del_cascade
AFTER DELETE ON Dept_tab
FOR EACH ROW

-- Before a row is deleted from Dept_tab, delete all
-- rows from the Emp_tab table whose DEPTNO is the same as
-- the DEPTNO being deleted from the Dept_tab table:
BEGIN
    DELETE FROM Emp_tab
        WHERE Emp_tab.Deptno = :old.Deptno;
END;
```

---



---

**Note:** Typically, the code for DELETE CASCADE is combined with the code for UPDATE SET NULL or UPDATE SET DEFAULT to account for both updates and deletes.

---



---

**UPDATE Cascade Trigger for Parent Table** The following trigger ensures that if a department number is updated in the Dept\_tab table, then this change is propagated to dependent foreign keys in the Emp\_tab table:

```
-- Generate a sequence number to be used as a flag for
-- determining if an update has occurred on a column:
CREATE SEQUENCE Update_sequence
    INCREMENT BY 1 MAXVALUE 5000
    CYCLE;

CREATE OR REPLACE PACKAGE Integritypackage AS
    Updateseq NUMBER;
END Integritypackage;

CREATE OR REPLACE PACKAGE BODY Integritypackage AS
END Integritypackage;
-- create flag col:
ALTER TABLE Emp_tab ADD Update_id NUMBER;

CREATE OR REPLACE TRIGGER Dept_cascade1 BEFORE UPDATE OF Deptno ON Dept_tab
DECLARE
    Dummy NUMBER;

-- Before updating the Dept_tab table (this is a statement
-- trigger), generate a new sequence number and assign
-- it to the public variable UPDATESEQ of a user-defined
-- package named INTEGRITYPACKAGE:
BEGIN
    SELECT Update_sequence.NEXTVAL
        INTO Dummy
        FROM dual;
    Integritypackage.Updateseq := Dummy;
END;

CREATE OR REPLACE TRIGGER Dept_cascade2 AFTER DELETE OR UPDATE
    OF Deptno ON Dept_tab FOR EACH ROW

-- For each department number in Dept_tab that is updated,
-- cascade the update to dependent foreign keys in the
```

```
-- Emp_tab table. Only cascade the update if the child row
-- has not already been updated by this trigger:
BEGIN
  IF UPDATING THEN
    UPDATE Emp_tab
      SET Deptno = :new.Deptno,
          Update_id = Integritypackage.Updateseq --from 1st
      WHERE Emp_tab.Deptno = :old.Deptno
      AND Update_id IS NULL;
      /* only NULL if not updated by the 3rd trigger
      fired by this same triggering statement */
  END IF;
  IF DELETING THEN

    -- Before a row is deleted from Dept_tab, delete all
    -- rows from the Emp_tab table whose DEPTNO is the same as
    -- the DEPTNO being deleted from the Dept_tab table:
    DELETE FROM Emp_tab
      WHERE Emp_tab.Deptno = :old.Deptno;
  END IF;
END;
CREATE OR REPLACE TRIGGER Dept_cascade3 AFTER UPDATE OF Deptno ON Dept_tab
BEGIN UPDATE Emp_tab
  SET Update_id = NULL
  WHERE Update_id = Integritypackage.Updateseq;
END;
```

---

**Note:** Because this trigger updates the Emp\_tab table, the Emp\_dept\_check trigger, if enabled, is also fired. The resulting mutating table error is trapped by the Emp\_dept\_check trigger. You should carefully test any triggers that require error trapping to succeed to ensure that they always work properly in your environment.

---

## Complex Check Constraints

Triggers can enforce integrity rules other than referential integrity. For example, this trigger performs a complex check before allowing the triggering statement to run.

---



---

**Note:** You may need to set up the following data structures for the example to work:

```
CREATE TABLE Salgrade (
  Grade          NUMBER,
  Losal          NUMBER,
  Hisal          NUMBER,
  Job_classification  NUMBER)
```

---



---

```
CREATE OR REPLACE TRIGGER Salary_check
BEFORE INSERT OR UPDATE OF Sal, Job ON Emp99
FOR EACH ROW
DECLARE
  Minsal          NUMBER;
  Maxsal          NUMBER;
  Salary_out_of_range  EXCEPTION;
BEGIN

  /* Retrieve the minimum and maximum salary for the
  employee's new job classification from the SALGRADE
  table into MINSAL and MAXSAL: */

  SELECT Minsal, Maxsal INTO Minsal, Maxsal FROM Salgrade
  WHERE Job_classification = :new.Job;

  /* If the employee's new salary is less than or greater
  than the job classification's limits, the exception is
  raised. The exception message is returned and the
  pending INSERT or UPDATE statement that fired the
  trigger is rolled back:*/

  IF (:new.Sal < Minsal OR :new.Sal > Maxsal) THEN
    RAISE Salary_out_of_range;
  END IF;
EXCEPTION
  WHEN Salary_out_of_range THEN
    Raise_application_error (-20300,
      'Salary '||TO_CHAR(:new.Sal)||' out of range for '
```

```
        || 'job classification ' || :new.Job
        || ' for employee ' || :new.Ename);
WHEN NO_DATA_FOUND THEN
    Raise_application_error(-20322,
        'Invalid Job Classification '
        || :new.Job_classification);
END;
```

## Complex Security Authorizations and Triggers

Triggers are commonly used to enforce complex security authorizations for table data. Only use triggers to enforce complex security authorizations that cannot be defined using the database security features provided with Oracle. For example, a trigger can prohibit updates to salary data of the `Emp_tab` table during weekends, holidays, and non-working hours.

When using a trigger to enforce a complex security authorization, it is best to use a `BEFORE` statement trigger. Using a `BEFORE` statement trigger has these benefits:

- The security check is done before the triggering statement is allowed to run, so that no wasted work is done by an unauthorized statement.
- The security check is performed only once for the triggering statement, not for each row affected by the triggering statement.

This example shows a trigger used to enforce security.

---

---

**Note:** You may need to set up the following data structures for the example to work:

```
CREATE TABLE Company_holidays (Day DATE);
```

---

---

```
CREATE OR REPLACE TRIGGER Emp_permit_changes
BEFORE INSERT OR DELETE OR UPDATE ON Emp99
DECLARE
    Dummy          INTEGER;
    Not_on_weekends EXCEPTION;
    Not_on_holidays EXCEPTION;
    Non_working_hours EXCEPTION;
BEGIN
    /* check for weekends: */
    IF (TO_CHAR(Sysdate, 'DY') = 'SAT' OR
        TO_CHAR(Sysdate, 'DY') = 'SUN') THEN
        RAISE Not_on_weekends;
    END IF;
```

```

/* check for company holidays:*/
SELECT COUNT(*) INTO Dummy FROM Company_holidays
WHERE TRUNC(Day) = TRUNC(Sysdate);
/* TRUNC gets rid of time parts of dates: */
IF dummy > 0 THEN
  RAISE Not_on_holidays;
END IF;
/* Check for work hours (8am to 6pm): */
IF (TO_CHAR(Sysdate, 'HH24') < 8 OR
    TO_CHAR(Sysdate, 'HH24') > 18) THEN
  RAISE Non_working_hours;
END IF;
EXCEPTION
WHEN Not_on_weekends THEN
  Raise_application_error(-20324,'May not change '
    ||'employee table during the weekend');
WHEN Not_on_holidays THEN
  Raise_application_error(-20325,'May not change '
    ||'employee table during a holiday');
WHEN Non_working_hours THEN
  Raise_application_error(-20326,'May not change '
    ||'Emp_tab table during non-working hours');
END;

```

## Transparent Event Logging and Triggers

Triggers are very useful when you want to transparently perform a related change in the database following certain events.

The REORDER trigger example shows a trigger that reorders parts as necessary when certain conditions are met. (In other words, a triggering statement is entered, and the PARTS\_ON\_HAND value is less than the REORDER\_POINT value.)

## Derived Column Values and Triggers

Triggers can derive column values automatically, based upon a value provided by an INSERT or UPDATE statement. This type of trigger is useful to force values in specific columns that depend on the values of other columns in the same row. BEFORE row triggers are necessary to complete this type of operation for the following reasons:

- The dependent values must be derived before the `INSERT` or `UPDATE` occurs, so that the triggering statement can use the derived values.
- The trigger must fire for each row affected by the triggering `INSERT` or `UPDATE` statement.

The following example illustrates how a trigger can be used to derive new column values for a table whenever a row is inserted or updated.

---

---

**Note:** You may need to set up the following data structures for the example to work:

```
ALTER TABLE Emp99 ADD(  
    Uppername   VARCHAR2(20),  
    Soundexname VARCHAR2(20));
```

---

---

```
CREATE OR REPLACE TRIGGER Derived  
BEFORE INSERT OR UPDATE OF Ename ON Emp99  
  
/* Before updating the ENAME field, derive the values for  
   the UPPERNAME and SOUNDEXNAME fields. Users should be  
   restricted from updating these fields directly: */  
FOR EACH ROW  
BEGIN  
    :new.Uppername := UPPER(:new.Ename);  
    :new.Soundexname := SOUNDEX(:new.Ename);  
END;
```

### Building Complex Updatable Views

Views are an excellent mechanism to provide logical windows over table data. However, when the view query gets complex, the system implicitly cannot translate the DML on the view into those on the underlying tables. `INSTEAD OF` triggers help solve this problem. These triggers can be defined over views, and they fire *instead* of the actual DML.

Consider a library system where books are arranged under their respective titles. The library consists of a collection of book type objects. The following example explains the schema.

```

CREATE OR REPLACE TYPE Book_t AS OBJECT
(
  Booknum  NUMBER,
  Title    VARCHAR2(20),
  Author   VARCHAR2(20),
  Available CHAR(1)
);
CREATE OR REPLACE TYPE Book_list_t AS TABLE OF Book_t;

```

Assume that the following tables exist in the relational schema:

Table Book\_table (Booknum, Section, Title, Author, Available)

Booknum	Section	Title	Author	Available
121001	Classic	Iliad	Homer	Y
121002	Novel	Gone With the Wind	Mitchell M	N

Library consists of library\_table(section).

### Section

Geography

Classic

Now you can define a complex view over these tables to create a logical view of the library with sections and a collection of books in each section.

```

CREATE OR REPLACE VIEW Library_view AS
SELECT i.Section, CAST (MULTISET (
  SELECT b.Booknum, b.Title, b.Author, b.Available
  FROM Book_table b
  WHERE b.Section = i.Section) AS Book_list_t) BOOKLIST
FROM Library_table i;

```

Make this view updatable by defining an INSTEAD OF trigger over the view.

```

CREATE OR REPLACE TRIGGER Library_trigger INSTEAD OF INSERT ON Library_view FOR
EACH ROW
  Bookvar BOOK_T;
  i      INTEGER;

```

```
BEGIN
  INSERT INTO Library_table VALUES (:NEW.Section);
  FOR i IN 1..:NEW.Booklist.COUNT LOOP
    Bookvar := Booklist(i);
    INSERT INTO book_table
      VALUES ( Bookvar.booknum, :NEW.Section, Bookvar.Title, Bookvar.Author,
bookvar.Available);
    END LOOP;
END;
/
```

Now, the `library_view` is an updatable view, and any INSERTs on the view are handled by the trigger that gets fired automatically. For example:

```
INSERT INTO Library_view VALUES ('History', book_list_t(book_t(121330,
'Alexander', 'Mirth', 'Y'));
```

Similarly, you can also define triggers on the nested table `booklist` to handle modification of the nested table element.

### Tracking System Events

**Fine-Grained Access Control** System triggers can be used to set application context. Application context is an Oracle8i feature which enhances your ability to implement fine-grained access control. Application context is a secure session cache, and it can be used to store session-specific attributes.

In the example that follows, procedure `set_ctx` sets the application context based on the user profile. The trigger `setexpensectx` ensures that the context is set for every user.

```
CONNECT secdemo/secdemo

CREATE OR REPLACE CONTEXT Expenses_reporting USING Secdemo.Exprep_ctx;

REM =====
REM Creation of the package which implements the context:
REM =====

CREATE OR REPLACE PACKAGE Exprep_ctx AS
  PROCEDURE Set_ctx;
END;

SHOW ERRORS
```

```

CREATE OR REPLACE PACKAGE BODY Exprep_ctx IS
  PROCEDURE Set_ctx IS
    Empnum    NUMBER;
    Countrec  NUMBER;
    Cc        NUMBER;
    Role      VARCHAR2(20);
  BEGIN

    -- SET emp_number:
    SELECT Employee_id INTO Empnum FROM Employee
       WHERE Last_name = SYS_CONTEXT('userenv', 'session_user');

    DBMS_SESSION.SET_CONTEXT('expenses_reporting', 'emp_number', Empnum);

    -- SET ROLE:
    SELECT COUNT (*) INTO Countrec FROM Cost_center WHERE Manager_id=Empnum;
    IF (countrec > 0) THEN
      DBMS_SESSION.SET_CONTEXT('expenses_reporting', 'exp_role', 'MANAGER');
    ELSE
      DBMS_SESSION.SET_CONTEXT('expenses_reporting', 'exp_role', 'EMPLOYEE');
    END IF;

    -- SET cc_number:
    SELECT Cost_center_id INTO Cc FROM Employee
       WHERE Last_name = SYS_CONTEXT('userenv', 'session_user');
    DBMS_SESSION.SET_CONTEXT('expenses_reporting', 'cc_number', Cc);
  END;
END;

```

**CALL Syntax**

```

CREATE OR REPLACE TRIGGER Secdemo.Setexpseetx
AFTER LOGON ON DATABASE
CALL Secdemo.Exprep_ctx.Set_ctx

```

## Triggering Event Publication

Oracle's system event publication lets applications subscribe to database events, just like they subscribe to messages from other applications.

**See Also:** [Chapter 13, "Working With System Events"](#)

Oracle's system events publication framework includes the following features:

- Infrastructure for publish/subscribe, by making the database an active publisher of events.
- Integration of data cartridges in the server: The system events publication can be used to notify cartridges of state changes in the server.
- Integration of fine-grained access control in the server.

## Publication Framework

The Oracle framework allows declarative definition of system event publication. This enables triggers to support database events, and users can specify a procedure that is to be run when the event occurs. DML events are supported on tables, and system events are supported on `DATABASE` and `SCHEMA`.

The system event publication subsystem tightly integrates with the Advanced Queuing publish/subscribe engine. The `DBMS_AQ.ENQUEUE()` procedure is used by publish/subscribe applications, and callouts are used by non-publish/subscribe applications, like cartridges.

Users or administrators can enable publication of system events by creating triggers specifying the publication attributes. By default, a trigger (and, therefore, publication of events specified in the trigger) is enabled. Users can also disable publication of these events by disabling the trigger, using the `ALTER TRIGGER` statement.

**See Also:** *Oracle8i SQL Reference*

For details on how to subscribe to published events and how to specify the delivery of these published events, see *Oracle8i Application Developer's Guide - Advanced Queuing* and *Oracle Call Interface Programmer's Guide*

## Event Publication

When events are detected by the server, the trigger mechanism executes the action specified in the trigger. As part of this action, you can use the `DEMS_AQ` package to publish the event to a queue, which then enables subscribers to get notifications.

---

---

**Note:** Detection of an event is predefined for a given release of the server. There is no user-specified event detection mechanism.

---

---

When an event occurs, all triggers that are enabled on that event are fired, with some exceptions:

- If the trigger is actually the target of the triggering event, it is not fired. For example, a trigger for all DROP events is not fired when it is dropped itself.
- If a trigger has been modified but not committed within the same transaction as the firing event, it is not fired. For example, recursive DDL within a system trigger might modify a trigger, which prevents the modified trigger from being fired by events within the same transaction.

More than one trigger can be created on an object; therefore, it is possible that more than one publication is made in response to the same event, and there should be no publication ordering assumptions. The publications are made in the order in which the system events transpire.

**Publication Context** When an event is published, certain runtime context and attributes, as specified in the parameter list, are passed to the callout procedure. A set of functions called event attribute functions are provided.

**See Also:** For event-specific attributes, see "[Event Attribute Functions](#)" on page 13-2.

For each system event supported, event-specific attributes are identified and predefined for the event. You can choose the parameter list to be any of these attributes, along with other simple expressions. For callouts, these are passed as `IN` arguments.

**Error Handling** Return status from publication callout functions for all events are ignored. For example, with `SHUTDOWN` events, the server cannot do anything with the return status.

**See Also:** For details on return status, see "[List of Events](#)" on page 13-6.

**Execution Model** Traditionally, triggers execute as the definer of the trigger. The trigger action of an event is executed as the definer of the action (as the definer of the package or function in callouts, or as owner of the trigger in queues). Because the owner of the trigger must have EXECUTE privileges on the underlying queues, packages, or procedure, this behavior is consistent.

---

## Working With System Events

System events, like `LOGON` and `SHUTDOWN`, provide a mechanism for tracking system changes. Oracle lets you combine this tracking with database event notification, which provides a simple and elegant method of delivering asynchronous messaging to an application.

This chapter includes descriptions of the various events on which triggers can be created. It also provides the list of event attribute functions.

**See Also:** To use the information in this chapter, you need to understand [Chapter 12, "Using Triggers"](#).

## Event Attribute Functions

You can obtain certain event-specific attributes when a trigger is fired. These attributes can be used as standalone functions.

### Usage Notes

- To make these attributes available, you must first run the CATPROC.SQL script.
- The trigger dictionary object maintains metadata about events that will be published and their corresponding attributes.
- In earlier releases, these functions were accessed through the SYS package. We recommend you use these public synonyms whose names begin with ora\_

**Table 13–1 System Defined Event Attributes**

Attribute	Type	Description	Example
ora_client_ip_address	VARCHAR2	Returns the IP address of the client in a LOGON event, when the underlying protocol is TCP/IP	<pre>if (ora_sysevent = 'LOGON')   then addr := ora_client_ip_ address; end if;</pre>
ora_database_name	VARCHAR2(50)	Database name.	<pre>DECLARE   db_name VARCHAR2(50); BEGIN   db_name := ora_database_name; END;</pre>
ora_des_encrypted_password	VARCHAR2	The DES encrypted password of the user being created or altered.	<pre>IF (ora_dict_obj_type = 'USER')   THEN INSERT INTO event_table (ora_des_encrypted_password); END IF;</pre>
ora_dict_obj_name	VARCHAR(30)	Name of the dictionary object on which the DDL operation occurred.	<pre>INSERT INTO event_table ('Changed object is '    ora_ dict_obj_name');</pre>

**Table 13–1 System Defined Event Attributes (Cont.)**

Attribute	Type	Description	Example
ora_dict_obj_name_list (name_list OUT ora_name_list_t)	BINARY_INTEGER	Return the list of object names of objects being modified in the event.	if (ora_sysevent = 'ASSOCIATE STATISTICS') then number_modified := ora_dict_obj_name_list (name_list); end if;
ora_dict_obj_owner	VARCHAR(30)	Owner of the dictionary object on which the DDL operation occurred.	INSERT INTO event_table ('object owner is'    ora_dict_obj_owner');
ora_dict_obj_owner_list(owner_list OUT ora_name_list_t)	BINARY_INTEGER	Returns the list of object owners of objects being modified in the event.	if (ora_sysevent = 'ASSOCIATE STATISTICS') then number_of_modified_objects := ora_dict_obj_owner_list(owner_list); end if;
ora_dict_obj_type	VARCHAR(20)	Type of the dictionary object on which the DDL operation occurred.	INSERT INTO event_table ('This object is a '    ora_dict_obj_type);
ora_grantee( user_list OUT ora_name_list_t)	BINARY_INTEGER	Returns the grantees of a grant event in the OUT parameter; returns the number of grantees in the return value.	if (ora_sysevent = 'GRANT') then number_of_users := ora_grantee(user_list); end if;
ora_instance_num	NUMBER	Instance number.	IF (ora_instance_num = 1) THEN INSERT INTO event_table ('1'); END IF;
ora_is_alter_column( column_name IN VARCHAR2)	BOOLEAN	Returns true if the specified column is altered.	if (ora_sysevent = 'ALTER' and ora_dict_obj_type = 'TABLE') then alter_column := ora_is_alter_column('FOO'); end if;

**Table 13–1 System Defined Event Attributes (Cont.)**

Attribute	Type	Description	Example
ora_is_creating_nested_table	BOOLEAN	Return TRUE if the current event is creating a nested table	if (ora_sysevent = 'CREATE' and ora_dict_obj_type = 'TABLE' and ora_is_creating_nested_table) then insert into event_tab values ('A nested table is created'); end if;
ora_is_drop_column( column_name IN VARCHAR2)	BOOLEAN	Returns true if the specified column is dropped.	if (ora_sysevent = 'ALTER' and ora_dict_obj_type = 'TABLE') then drop_column := ora_is_drop_column('FOO'); end if;
ora_is_servererror	BOOLEAN	Returns TRUE if given error is on error stack, FALSE otherwise.	IF (ora_is_servererror(error_number)) THEN INSERT INTO event_table ('Server error!!'); END IF;
ora_login_user	VARCHAR2(30)	Login user name.	SELECT ora_login_user FROM dual;
ora_privileges( privilege_list OUT ora_name_list_t)	BINARY_INTEGER	Returns the list of privileges being granted by the grantee or the list of privileges revoked from the revokee in the OUT parameter; returns the number of privileges in the return value.	if (ora_sysevent = 'GRANT' or ora_sysevent = 'REVOKE') then number_of_privileges := ora_privileges(priv_list); end if;

**Table 13–1 System Defined Event Attributes (Cont.)**

Attribute	Type	Description	Example
<code>ora_revokee (user_list OUT ora_name_list_t)</code>	BINARY_INTEGER	Returns the revokees of a revoke event in the OUT parameter; returns the number of revokees in the return value..	<pre>if (ora_sysevent = 'REVOKE') then   number_of_users := ora_revokee(user_list);</pre>
<code>ora_server_error</code>	NUMBER	Given a position(1 for top of stack), it returns the error number at that position on error stack	<pre>INSERT INTO event_table ('top stack error '    ora_server_error(1));</pre>
<code>ora_sysevent</code>	VARCHAR2(20)	System event firing the trigger: Event name is same as that in the syntax.	<pre>INSERT INTO event_table (ora_sysevent);</pre>
<code>ora_with_grant_option</code>	BOOLEAN	Returns true if the privileges are granted with grant option.	<pre>if (ora_sysevent = 'GRANT' and ora_with_grant_option = TRUE)   then insert into event_table ('with grant option'); end if;</pre>

## List of Events

### Resource Manager Events

Resource manager events are related to instance startup and shutdown. Triggers created on resource manager events must be associated with the database object.

Table 13-2 contains a list of resource manager events.

**Table 13-2 Resource Manager Events**

Event	When Fired?	Conditions	Restrictions	Transaction	Attribute Functions
STARTUP	When the database is opened.	None allowed	No database operations allowed Return status ignored.	Starts a separate transaction and commits it after firing the triggers.	ora_sysevent ora_login_user ora_instance_num ora_database_name
SHUTDOWN	Just before the server starts the shutdown of an instance.  This lets the cartridge shutdown completely. For abnormal instance shutdown, this event may not be fired.	None allowed	No database operations allowed Return status ignored.	Starts a separate transaction and commits it after firing the triggers.	ora_sysevent ora_login_user ora_instance_num ora_database_name
SERVERERROR	When the error <code>eno</code> occurs. If no condition is given, then this event fires when any error occurs.  Does not apply to ORA-1034, ORA-1403, ORA-1422, ORA-1423, and ORA-4030 conditions, because they are not true errors or are too serious to continue processing.	ERRNO = <code>eno</code>	Depends on the error. Return status ignored.	Starts a separate transaction and commits it after firing the triggers.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_server_error ora_is_servererror

## Client Events

Client events are the events related to user logon/logoff, DML, and DDL operations. For example:

```
CREATE OR REPLACE TRIGGER On_Logon
  AFTER LOGON
  ON The_user.Schema
BEGIN
  Do_Something;
END;
```

The LOGON and LOGOFF events allow simple conditions on UID() and USER(). All other events allow simple conditions on the type and name of the object, as well as functions like UID() and USER().

The LOGON event starts a separate transaction and commits it after firing the triggers. All other events fire the triggers in the existing user transaction.

The LOGON and LOGOFF events can operate on any objects. For all other events, the corresponding trigger cannot perform any DDL operations, such as DROP and ALTER, on the object that caused the event to be generated.

The DDL allowed inside these triggers is altering, creating, or dropping a table, creating a trigger, and compile operations.

If an event trigger becomes the target of a DDL operation (such as CREATE TRIGGER), it cannot be fired later during the same transaction

[Table 13-3](#) contains a list of client events.

**Table 13–3 Client Events**

<b>Event</b>	<b>When Fired?</b>	<b>Attribute Functions</b>
AFTER LOGON	After a successful logon of a user.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_client_ip_address
BEFORE LOGOFF	At the start of a user logoff	ora_sysevent ora_login_user ora_instance_num ora_database_name
BEFORE CREATE AFTER CREATE	When a catalog object is created.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_type ora_dict_obj_name ora_dict_obj_owner ora_is_creating_nested_table (for CREATE TABLE events)
BEFORE ALTER AFTER ALTER	When a catalog object is altered.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_type ora_dict_obj_name ora_dict_obj_owner ora_des_encrypted_password (for ALTER USER events) ora_is_alter_column, ora_is_drop_column (for ALTER TABLE events)
BEFORE DROP AFTER DROP	When a catalog object is dropped.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_type ora_dict_obj_name ora_dict_obj_owner

**Table 13–3 Client Events**

<b>Event</b>	<b>When Fired?</b>	<b>Attribute Functions</b>
BEFORE ANALYZE AFTER ANALYZE	When an analyze statement is issued	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner
BEFORE ASSOCIATE STATISTICS AFTER ASSOCIATE STATISTICS	When an associate statistics statement is issued	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_dict_obj_name_list ora_dict_obj_owner_list
BEFORE AUDIT AFTER AUDIT  BEFORE NOAUDIT AFTER NOAUDIT	When an audit or noaudit statement is issued	ora_sysevent ora_login_user ora_instance_num ora_database_name
BEFORE COMMENT AFTER COMMENT	When an object is commented	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner
BEFORE CREATE AFTER CREATE	When an object is created.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_is_creating_nested_table (for CREATE TABLE events)

**Table 13–3 Client Events**

Event	When Fired?	Attribute Functions
BEFORE DDL AFTER DDL	When most SQL DDL statements are issued. Not fired for ALTER DATABASE, CREATE CONTROLFILE, CREATE DATABASE, and DDL issued through the PL/SQL procedure interface, such as creating an advanced queue.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner
BEFORE DISASSOCIATE STATISTICS AFTER DISASSOCIATE STATISTICS	When a disassociate statistics statement is issued	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_dict_obj_name_list ora_dict_obj_owner_list
BEFORE GRANT AFTER GRANT	When a grant statement is issued	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_grantee ora_with_grant_option ora_privileges

**Table 13–3 Client Events**

<b>Event</b>	<b>When Fired?</b>	<b>Attribute Functions</b>
BEFORE RENAME AFTER RENAME	When a rename statement is issued.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_owner ora_dict_obj_type
BEFORE REVOKE AFTER REVOKE	When a revoke statement is issued	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_revokee ora_privileges
BEFORE TRUNCATE AFTER TRUNCATE	When an object is truncated	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner



---

## Using Publish-Subscribe

Because the database is the most significant resource of information within the enterprise, Oracle created a publish-subscribe solution for enterprise information delivery and messaging to complement this role. Topics in this chapter include:

- [Introduction to Publish-Subscribe](#)
- [Publish-Subscribe Infrastructure](#)
- [Publish-Subscribe Concepts](#)
- [Examples](#)

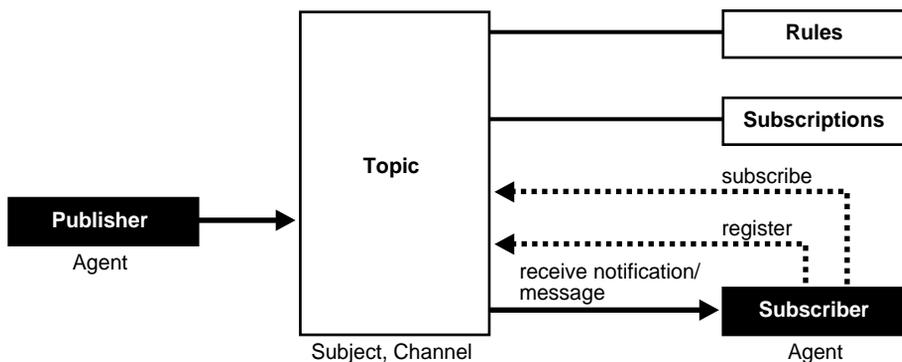
## Introduction to Publish-Subscribe

Networking technologies and products now enable a high degree of connectivity across a large number of computers, applications, and users. In these environments, it is important to provide asynchronous communications for the class of distributed systems that operate in a loosely-coupled and autonomous fashion, and which require operational immunity from network failures. This requirement has been filled by various middleware products that are characterized as messaging, message oriented middleware (MOM), message queuing, or publish-subscribe.

Applications that communicate through a publish and subscribe paradigm require the sending applications (publishers) to publish messages without explicitly specifying recipients or having knowledge of intended recipients. Similarly, receiving applications (subscribers) must receive only those messages that the subscriber has registered an interest in.

This decoupling between senders and recipients is usually accomplished by an intervening entity between the publisher and the subscriber, which serves as a level of indirection. This intervening entity is a queue that represents a subject or channel. [Figure 14-1](#) illustrates publish and subscribe functionality.

**Figure 14-1 Oracle Publish-Subscribe Functionality**



A subscriber subscribes to a queue by expressing interest in messages enqueued to that queue and by using a subject- or content-based rule as a filter. This results in a set of rule-based subscriptions associated with a given queue.

At runtime, publishers post messages to various queues. The queue (in other words, the delivery mechanisms of the underlying infrastructure) then delivers messages that match the various subscriptions to the appropriate subscribers.

## Publish-Subscribe Infrastructure

Oracle includes the following infrastructure and features to support database-enabled publish-subscribe messaging:

- [Database Events](#)
- [Advanced Queuing](#)
- [Client Notifications](#)

### Database Events

Database events support declarative definitions for publishing database events, detection, and run-time publication of such events. This feature enables active publication of information to end-users in an event-driven manner, to complement the traditional pull-oriented approaches to accessing information.

**See Also:** [Chapter 13, "Working With System Events"](#)

### Advanced Queuing

Oracle Advanced Queuing supports a queue-based publish-subscribe paradigm. Database queues serve as a durable store for messages, along with capabilities to allow publish and subscribe based on queues. A rules-engine and subscription service dynamically route messages to recipients based on expressed interest. This allows decoupling of addressing between senders and receivers to complement the existing explicit sender-receiver message addressing.

**See Also:** *Oracle8i Application Developer's Guide - Advanced Queuing*

### Client Notifications

Client notifications support asynchronous delivery of messages to interested subscribers. This enables database clients to register interest in certain queues, and it enables these clients to receive notifications when publications on such queues occur. Asynchronous delivery of messages to database clients is in contrast to the traditional polling techniques used to retrieve information.

**See Also:** *Oracle Call Interface Programmer's Guide*

## Publish-Subscribe Concepts

This section describes various concepts related to publish-subscribe.

### Queues

A queue is an entity that supports the notion of named subjects of interest. Queues can be characterized as:

**Non-Persistent Queues (Lightweight Queues)** The underlying queue infrastructure pushes the messages published to connected clients in a lightweight, at-best-once, manner.

**Persistent Queues** Queues serve as durable containers for messages. Messages are delivered in a deferred and reliable mode.

### Agent

Publishers and subscribers are internally represented as agents. There is a distinction between an agent and a client.

An *agent* is a persistent logical subscribing entity that expresses interest in a queue via a subscription. An agent has properties, such as an associated subscription, an address, and a delivery mode for messages. In this context, an agent is an electronic proxy for a publisher or subscriber.

A *client* is a transient physical entity. The attributes of a client include the physical process where the client programs run, the node name, and the client application logic. There could be several clients acting on behalf of a single agent. Also, the same client, if authorized, can act on behalf of multiple agents.

### Rules

A rule on a queue is specified as a conditional expression using a predefined set of operators on the message format attributes or on the message header attributes. Each queue has an associated message content format that describes the structure of the messages represented by that queue. The message format may be unstructured (RAW) or it may have a well-defined structure (ADT). This allows both subject- or content-based subscriptions.

## Subscriber

Subscribers (agents) may specify subscriptions on a queue using a rule. Subscribers are durable and are stored in a catalog.

## Database Event Publication Framework

The database represents a significant source for publishing information. An event framework is proposed to allow declarative definition of database event publication. As these pre-defined events occur, the framework detects and publishes such events. This allows active delivery of information to end-users in an event-driven manner as part of the publish-subscribe capability.

## Registration

Registration is the process of associated delivery information by a given client, acting on behalf of an agent. There is an important distinction between the subscription and registration related to the agent/client separation.

Subscription indicates an interest in a particular queue by an agent. It does not specify where and how delivery must occur. Delivery information is a physical property that is associated with a client, and it is a transient manifestation of the logical agent; i.e., the subscriber. A specific client process acting on behalf of an agent registers delivery information by associating a host and port, indicating *where* the delivery should be done, and a callback, indicating *how* there delivery should be done.

## Publish Message

Publishers publish messages to queues by using the appropriate queuing interfaces. The interfaces may depend on which model the queue is implemented on. For example, an enqueue call represents the publishing of a message.

## Rules Engine

When a message is posted or published to a given queue, a rules engine extracts the set of candidate rules from all rules defined on that queue that match the published message.

## Subscription Services

Corresponding to the list of candidate rules on a given queue, the set of subscribers that match the candidate rules can be evaluated. In turn, the set of agents corresponding to this subscription list can be determined and notified.

## Posting

The queue notifies all registered clients of the appropriate published messages. This concept is called *posting*. When the queue needs to notify all interested clients, it posts the message to all registered clients.

## Receive Message

A subscriber may receive messages via any of the following mechanisms:

- A client process acting on behalf of the subscriber specifies a callback using the registration mechanism. The posting mechanism then asynchronously invokes the callback when a message matches the subscriber's subscription. The message content may be passed to the callback function (non-persistent queues only).
- A client process acting on behalf of the subscriber specifies a callback using the registration mechanism. The posting mechanism then asynchronously invokes the callback function, but without the full message content. This serves as a notification to the client, which subsequently retrieves the message content in a pull fashion (persistent queues only).
- A client process acting on behalf of the subscriber simply retrieves messages from the queue in a periodic, or some other appropriate, manner. While the messages are deferred, there is no asynchronous delivery to the end-client.

## Examples

---

---

**Note:** You may need to set up data structures, similar to the following, for certain examples to work:

```
CONNECT system/manager
DROP USER pubsub CASCADE;
CREATE USER pubsub IDENTIFIED BY pubsub;
GRANT CONNECT, RESOURCE TO pubsub;
GRANT EXECUTE ON DBMS_AQ TO pubsub;
GRANT EXECUTE ON DBMS_AQADM TO pubsub;
GRANT AQ_ADMINISTRATOR_ROLE TO pubsub;
CONNECT pubsub/pubsub
```

---

---

Scenario: This example shows how system events, client notification, and AQ work together to implement publish-subscribe.

- Create under the user schema, `pubsub`, with all objects necessary to support a publish-subscribe mechanism. In this particular code, the Agent `snoop` subscribe to messages that are published at logon events. Note that the user `pubsub` needs `AQ_ADMINISTRATOR_ROLE` privileges to use AQ functionalities.

```

■
Rem -----
Rem create queue table for persistent multiple consumers:
Rem -----

CONNECT pubsub/pubsub;

Rem Create or replace a queue table
BEGIN
DBMS_AQADM.CREATE_QUEUE_TABLE(
  Queue_table      => 'Pubsub.Raw_msg_table',
  Multiple_consumers => TRUE,
  Queue_payload_type => 'RAW',
  Compatible       => '8.1');
END;
/
Rem -----
Rem Create a persistent queue for publishing messages:
Rem -----

Rem Create a queue for logon events
begin
BEGIN
  DBMS_AQADM.CREATE_QUEUE(
    Queue_name      => 'Pubsub.Logon',
    Queue_table     => 'Pubsub.Raw_msg_table',
    Comment         => 'Q for error triggers');
END;
/

Rem -----
Rem Start the queue:
Rem -----

BEGIN
  DBMS_AQADM.START_QUEUE('pubsub.logon');
END;
/

```

```
Rem -----
Rem  define new_enqueue for convenience:
Rem -----

CREATE OR REPLACE PROCEDURE New_enqueue(
            Queue_name      IN VARCHAR2,
            Payload          IN RAW ,
            Correlation      IN VARCHAR2 := NULL,
            Exception_queue  IN VARCHAR2 := NULL)
AS

Enq_ct      DBMS_AQ.Enqueue_options_t;
Msg_prop   DBMS_AQ.Message_properties_t;
Enq_msgid  RAW(16);
Userdata   RAW(1000);

BEGIN
    Msg_prop.Exception_queue := Exception_queue;
    Msg_prop.Correlation := Correlation;
    Userdata := Payload;

DBMS_AQ.ENQUEUE(Queue_name, Enq_ct, Msg_prop, Userdata, Enq_msgid);
END;
/

Rem -----
Rem  add subscriber with rule based on current user name,
Rem  using correlation_id
Rem -----

DECLARE
Subscriber Sys.Aq$_agent;
BEGIN
    Subscriber := sys.aq$_agent('SNOOP', NULL, NULL);
DBMS_AQADM.ADD_SUBSCRIBER(
    Queue_name      => 'Pubsub.logon',
    Subscriber      => subscriber,
    Rule            => 'CORRID = ''SCOTT'' ');
END;
/

Rem -----
Rem  create a trigger on logon on database:
Rem -----
```

```

Rem create trigger on after logon:
CREATE OR REPLACE TRIGGER pubsub.Systrig2
  AFTER LOGON
  ON DATABASE
  BEGIN
    New_enqueue('Pubsub.Logon', HEXTORAW('9999'), Dbms_standard.login_user);
  END;
/

```

- After subscriptions are created, the next step is for the client to register for notification using callback functions. This is done using the Oracle Call Interface (OCI). The code below performs necessary steps for registration. The initial steps of allocating and initializing session handles are omitted here for sake of clarity.

```

ub4 namespace = OCI_SUBSCR_NAMESPACE_AQ;

/* callback function for notification of logon of user 'scott' on database: */

ub4 notifySnoop(ctx, subscrhp, pay, payl, desc, mode)
dvoid *ctx;
OCISubscription *subscrhp;
dvoid *pay;
ub4 payl;
dvoid *desc;
ub4 mode;
{
  printf("Notification : User Scott Logged on\n");
}

int main()
{
  OCISession *authp = (OCISession *) 0;
  OCISubscription *subscrhpSnoop = (OCISubscription *)0;

  /*****
    Initialize OCI Process/Environment
    Initialize Server Contexts
    Connect to Server
    Set Service Context
  *****/

  /* Registration Code Begins */

```

```
/* Each call to initSubscriptionHn allocates
   and Initialises a Registration Handle */

initSubscriptionHn(    &subscrhpSnoop,    /* subscription handle */
    "ADMIN:PUBSUB.SNOOP", /* subscription name */
    /* <agent_name>:<queue_name> */
    (dvoid*)notifySnoop); /* callback function */

/*****
   The Client Process does not need a live Session for Callbacks
   End Session and Detach from Server
   *****/

OCISessionEnd ( svchp, errhnp, authnp, (ub4) OCI_DEFAULT);

/* detach from server */
OCIserverDetach( srvhnp, errhnp, OCI_DEFAULT);

while (1)    /* wait for callback */
    sleep(1);
}

void initSubscriptionHn (subscrhp,
subscriptionName,
func)

OCISubscription **subscrhp;
char* subscriptionName;
dvoid * func;
{

/* allocate subscription handle: */

(void) OCIHandleAlloc((dvoid *) envhnp, (dvoid **)subscrhp,
    (ub4) OCI_HTYPE_SUBSCRIPTION,
    (size_t) 0, (dvoid **) 0);

/* set subscription name in handle: */

(void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
    (dvoid *) subscriptionName,
    (ub4) strlen((char *)subscriptionName),
```

```
(ub4) OCI_ATTR_SUBSCR_NAME, errhp);

/* set callback function in handle: */

(void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
    (dvoid *) func, (ub4) 0,
    (ub4) OCI_ATTR_SUBSCR_CALLBACK, errhp);

(void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
    (dvoid *) 0, (ub4) 0,
    (ub4) OCI_ATTR_SUBSCR_CTX, errhp);

/* set namespace in handle: */

(void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
    (dvoid *) &namespace, (ub4) 0,
    (ub4) OCI_ATTR_SUBSCR_NAMESPACE, errhp);

checkerr(errhp, OCISubscriptionRegister(svchp, subscrhp, 1, errhp,
    OCI_DEFAULT));
}
```

Now, if user SCOTT logged on to the database, the client is notified, and the call back function `notifySnoop` is called.



# Part IV

---

## Developing Specialized Applications

This part contains:

- [Chapter 15, "Developing Web Applications with PL/SQL"](#)
- [Chapter 16, "Working with Transaction Monitors with Oracle XA"](#)



---

## Developing Web Applications with PL/SQL

If you think that only new languages such as Java and Javascript can do network operations and produce dynamic web content, think again. PL/SQL has a number of features that you can use to web-enable your database and make your back-office data interactive and accessible to intranet users or even your customers.

### Performing Network Operations with PL/SQL

While PL/SQL's built-in features are focused on traditional database operations and programming logic, Oracle supplies packages that open up Internet computing to PL/SQL programmers. You can find details and examples using most of these packages in the *Oracle8i Supplied PL/SQL Packages Reference*.

#### Sending Mail

You can send mail from a PL/SQL program or stored procedure using the UTL\_SMTP package.

#### Getting a Host Name or Address

You can determine the hostname of the local machine, or the IP address of a given hostname from a PL/SQL program or stored procedure using the UTL\_INADDR package. You use the results in calls to the UTL\_TCP package.

#### Working with TCP/IP Connections

You can open TCP/IP connections to machines on the network, and read or write to the corresponding sockets, using the UTL\_TCP package.

## Retrieving the Contents of an HTTP URL

You can retrieve the contents of an HTTP URL using the `UTL_HTTP` package. The contents are typically in the form of HTML-tagged text, but may be plain text, a JPEG image, or any sort of file that is downloadable from a web server.

## Working with Tables, Image Maps, Cookies, CGI Variables, and More

Packages for all of these functions are supplied with the Oracle web gateways, such as Oracle Application Server (OAS) and WebDB. You can format the results of a query in an HTML table, produce an image map, set and get HTTP cookies, check the values of CGI variables, and combine other typical web operations with a PL/SQL program.

For details, refer to the PL/SQL Web Toolkit documentation that comes with the web gateways.

## Embedding PL/SQL Code in Web Pages (PL/SQL Server Pages)

To include dynamic content, including the results of SQL queries, inside web pages, you can use server-side scripting through *PL/SQL Server Pages* (PSP). You can author the web pages in a script-friendly HTML authoring tool, and drop the pieces of PL/SQL code into place. For cutting-edge web pages, you might find this technique more convenient than using the HTP and HTF packages to write out HTML content line by line.

Because the processing is done on the server -- in this case, the database server rather than the web server -- the browser receives a plain HTML page with no special script tags, and you can support all browsers and browser levels equally. It also makes network traffic efficient by minimizing the number of server roundtrips.

Embedding the PL/SQL code in the HTML page that you create lets you write content quickly and follow a rapid, iterative development process. You maintain central control of the software, with only a web browser required on the client machine.

The steps to implement a web-based solution using PL/SQL server pages are:

- [Choosing a Software Configuration](#)
- [Writing the Code and Content for the PL/SQL Server Page](#)
- [Loading the PL/SQL Server Page into the Database as a Stored Procedure](#)

## Choosing a Software Configuration

To develop and deploy PL/SQL Server Pages, you need the Oracle server at version 8.1.6 or later, together with a PL/SQL web gateway. Currently, the web gateways are the WebDB PL/SQL Gateway and the OAS PL/SQL Cartridge. Before you start with PSP, you should have access to both the database server and the web server for one of these gateways.

### Choosing Between PSP and the PL/SQL Web Toolkit

You can produce the same results in different ways:

- By writing an HTML page with embedded PL/SQL code and compiling it as a PL/SQL server page. You may call procedures from the PL/SQL Web Toolkit, but not to generate every line of HTML output.
- By writing a complete stored procedure that produces HTML by calling the HTP and OWA\_\* packages in the PL/SQL Web Toolkit.

The key factors in choosing between these techniques are:

- What source are you using as a starting point?
  - If you have a large body of HTML, and want to include dynamic content or make it the front end of a database application, use PSP.
  - If you have a large body of PL/SQL code that produces formatted output, you may find it more convenient to produce HTML tags by changing your print statements to call the HTP package of the PL/SQL Web Toolkit.
- What is the fastest and most convenient authoring environment for your group?
  - If most work is done using HTML authoring tools, use PSP.
  - If you use authoring tools that produce PL/SQL code for you, such as the page-building wizards in WebDB, then it might be less convenient to use PSP.

### How PSP Relates to Other Scripting Solutions

Because any kind of tags can be passed unchanged to the browser through a PL/SQL server page, you can include Javascript or other client-side script code in a PL/SQL server page.

You cannot mix PL/SQL server pages with other server-side script features, such as server-side includes. In many cases, you can get the same results by using the corresponding PSP features.

PSP uses the same script tag syntax as Java Server Pages (JSP), to make it easy to switch back and forth.

PSP uses syntax similar to that of Active Server Pages (ASP), although the syntax is not identical and you must typically translate from VBScript or JScript to PL/SQL. The best candidates for migration are pages that use the Active Data Object (ADO) interface to do database operations.

## Writing the Code and Content for the PL/SQL Server Page

You can start with an existing web page, or with an existing stored procedure. Either way, with a few additions and changes you can create dynamic web pages that perform database operations and display the results.

### The Format of the PSP File

The file for a PL/SQL server page must have the extension `.psp`.

It can contain whatever content you like, with text and tags interspersed with PSP directives, declarations, and scriptlets:

- In the simplest case, it is nothing more than an HTML file. Compiling it as a PL/SQL server page produces a stored procedure that outputs the exact same HTML file.
- In the most complex case, it is a PL/SQL procedure that generates all the content of the web page, including the tags for title, body, and headings.
- In the typical case, it is a mix of HTML (providing the static parts of the page) and PL/SQL (filling in the dynamic content).

The order and placement of the PSP directives and declarations is not significant in most cases -- only when another file is being included. For ease of maintenance, we recommend placing the directives and declarations together near the beginning of the file.

The following sections discuss the way to produce various results using the PSP scripting elements. If you are familiar with dynamic HTML and want to start coding right away, you can jump forward to [and](#) [.](#)

## Specifying the Scripting Language

To identify a file as a PL/SQL Server Page, include a

`<%@ page language="PL/SQL" %>` directive somewhere in the file. This directive is for compatibility with other scripting environments.

## Accepting User Input

User input comes encoded in the URL that retrieves the HTML page. You can generate the URL by hardcoding it in an HTML link, or by calling your page as the "action" of an HTML form. Your page receives the input as parameters to a PL/SQL stored procedure.

To set up parameter passing for a PL/SQL server page, include a

`<%@ plsql parameter="..." %>` directive. By default, parameters are of type `VARCHAR2`. To use a different type, include a `type="..."` attribute within the directive. To set a default value, so that the parameter becomes optional, include a `default="..."` attribute in the directive. The values for this attribute are substituted directly into a PL/SQL statement, so any strings must be single-quoted, and you can use special values such as `null`.

## Displaying HTML

The PL/SQL parts of the page are enclosed within special delimiters. All other content is passed along verbatim -- including any whitespace -- to the browser. To display text or HTML tags, write it as you would a typical web page. You do not need to call any output function.

Sometimes you might want to display one line of output or another, or change the value of an attribute, based on some condition. You can include `IF/THEN` logic and variable substitution inside the PSP delimiters, as shown in subsequent sections.

## Returning XML, Text, or Other Document Types

By default, the PL/SQL gateway transmits files as HTML documents, so that the browser formats them according to the HTML tags. If you want the browser to interpret the document as XML, plain text (with no formatting), or some other document type, include a `<%@ page contentType="..." %>` directive. (The attribute name is case-sensitive, so be sure to capitalize it as `contentType`.) Specify `text/html`, `text/xml`, `text/plain`, `image/jpeg`, or some other MIME type that the browser or other client program recognizes. Users may have to configure their browsers to recognize some MIME types.

Typically, a PL/SQL server page is intended to be displayed in a web browser. It could also be retrieved and interpreted by a program that can make HTTP requests, such as a Java or Perl application.

### **Returning Pages Containing Different Character Sets**

By default, the PL/SQL gateway transmits files using the character set defined by the web gateway. To convert the data to a different character set for displaying in a browser, include a `<%@ page charset="..." %>` directive. Specify Shift\_JIS, Big5, UTF-8, or other encoding that the browser or other client program recognizes.

You must also configure the character set setting in the database accessor descriptor (DAD) of the web gateway. Users may have to select the same encoding in their browsers to see the data displayed properly.

For example, a database in Japan may contain data in the EUC encoding, while the web browsers are set up to display Shift\_JIS encoding.

### **Handling Script Errors**

Any errors in HTML tagging are handled by the browser. The PSP loading process does not check for them.

If you make a syntax error in the PL/SQL code, the loader stops and you must fix the error before continuing. Note that any previous version of the stored procedure can be erased when you attempt to replace it and the script contains a syntax error. You might want to use one database for prototyping and debugging, then load the final stored procedure into a different database for production. You can switch databases using a command-line flag, without changing any source code.

To handle database errors that occur when the script runs, you can include PL/SQL exception-handling code within a PSP file, and have any unhandled exceptions bring up a special page. The page for unhandled exceptions is another PL/SQL server page with extension `.psp`. The error procedure does not receive any parameters, so to determine the cause of the error, it can call the `SQLCODE` and `SQLERRM` functions.

You can also display a standard HTML page without any scripting when an error occurs, but you must still give it the extension `.psp` and load it into the database as a stored procedure.

### **Naming the PL/SQL Stored Procedure**

Each top-level PL/SQL server page corresponds to a stored procedure within the server. By default, the procedure is given the same name as the original file, with

the .psp extension removed. To name the procedure something else, include a `<%@ page procedure="..." %>` directive.

### **Including the Contents of Other Files**

You can set up an include mechanism to pull in the contents of other files, typically containing either static HTML content or more PL/SQL scripting code. Include a `<%@ include file="..." %>` directive at the point where the other file's content should appear. Because the files are processed at the point where you load the stored procedure into the database, the substitution is done only once, not whenever the page is served.

You can use any names and extensions for the included files. If the included files contain PL/SQL scripting code, they do not need their own set of directives to identify the procedure name, character set, and so on.

When specifying the names of files to the PSP loader, you must include the names of all included files also. Specify the names of included files before the names of any .psp files.

You can use this feature to pull in the same content, such as a navigation banner, into many different files. Or, you can use it as a macro capability to include the same section of script code in more than one place in a page.

### **Declaring Variables for Use in the Script**

If you need to use global variables within the script, you can include a declaration block inside the delimiters `<%! %>`. All the usual PL/SQL syntax is allowed within the block. The delimiters serve as shorthand, letting you omit the DECLARE keyword. All the declarations are available to the code later on in the file.

You can specify multiple declaration blocks; internally, they are all merged into a single block when the PSP file is made into a stored procedure.

You can also use explicit DECLARE blocks within the `<% %>` delimiters that are explained later. These declarations are only visible to the following BEGIN/END block.

### **Specifying Statements to be Executed in the Script**

You can include any PL/SQL statements within the delimiters `<% %>`. The statements can be complete, or clauses of a compound statement, such as the IF part of an IF-THEN-ELSE statement. Any variables declared within DECLARE blocks are only visible to the following BEGIN/END block.

### **Substituting an Expression Result in a Script**

To include a value that depends upon the result of a PL/SQL expression, include the expression within the delimiters `<%= %>`. Because the result is always substituted in the middle of text or tags, it must be a string value or be able to be cast to a string. For any types that cannot be implicitly casted, such as DATE, pass the value to the PL/SQL `TO_CHAR` function.

The content between the `<%= %>` delimiters is processed by the `HTM.PRN` function, which trims any leading or trailing whitespace and requires that you quote any literal strings.

### **Conventions for Quoting and Escaping Strings**

When values specified in PSP attributes are used for PL/SQL operations, they are passed exactly as you specify them in the PSP file. If PL/SQL requires a single-quoted string, you must specify the string with the single quotes around it -- and surround the whole thing with double quotes.

You can also nest single-quoted strings inside single quotes. In this case, you must *escape* the nested single quotes by specifying the sequence `\'`.

Most characters and character sequences can be included in a PSP file without being changed by the PSP loader. To include the sequence `%>`, specify the escape sequence `%\>`. To include the sequence `<%`, specify the escape sequence `<\%`.

### **Retrieving a Result Set from a Query**

To retrieve a single row from the database, you can open a cursor, fetch data into it, and retrieve the columns from the cursor variable:

To display the results of a query that returns multiple rows, you can iterate through each row of the result set:

If you want to print out an entire table, you can call the `OWA_UTIL.TABLEPRINT` procedure from the PL/SQL web toolkit:

### **Updating the Database Based Upon User Input**

You can run insert, update, and delete operations within a PL/SQL server page. As with any program that is expected to return results as HTML, such pages should include some output to confirm that the operation is successful or to show the updated state.

## Coding Tips for PL/SQL Server Pages

To share procedures, constants, and types across different PL/SQL server pages, compile them into a separate package in the database (not using the PSP technique).

To make things easier to maintain, keep all your directives and declarations together near the beginning of a PL/SQL server page.

## Syntax of PL/SQL Server Page Elements

You can find examples of many of these elements in ["Examples of PL/SQL Server Pages"](#) on page 15-12.

### Page Directive

Specifies characteristics of the PL/SQL server page:

- What scripting language it uses.
- What type of information (MIME type) it produces.
- What other PSP file handles all uncaught exceptions.

Note that the attribute names `contentType` and `errorPage` are case-sensitive.

### Syntax

```
<%@ page [language="PL/SQL"] [contentType="content type string"] [errorPage="file.psp"] %>
```

### Procedure Directive

Specifies the name of the stored procedure produced by the PSP file. By default, the name is the filename without the `.psp` extension.

### Syntax

```
<%@ plsql procedure="procedure name" %>
```

### Parameter Directive

Specifies the name, and optionally the type and default, for each parameter expected by the PSP stored procedure. The parameters are passed using the CGI protocol, typically from an HTML form.

### Syntax

```
<%@ plsql parameter="parameter name" [type="PL/SQL type"] [default="value"] %>
```

### **Include Directive**

Specifies the name of a file to be included at a specific point in the PSP file. The file must have an extension other than .psp. It can contain HTML, PSP script elements, or a combination of both. The name resolution and file inclusion happens when the PSP file is loaded into the database as a stored procedure, so any changes to the file after that are not reflected when the stored procedure is run.

### **Syntax**

```
<%@ include file="path name" %>
```

### **Declaration Block**

Declares a set of PL/SQL variables that are visible throughout the page, not just within the next BEGIN/END block. This element typically spans multiple lines, with individual PL/SQL variable declarations ended by semicolons.

### **Syntax**

```
<%! PL/SQL declaration;  
    [ PL/SQL declaration; ] ... %>
```

### **Code Block (Scriptlet)**

Executes a set of PL/SQL statements when the stored procedure is run. This element typically spans multiple lines, with individual PL/SQL statements ended by semicolons. The statements can include complete blocks, or can be the bracketing parts of IF/THEN/ELSE or BEGIN/END blocks. When a code block is split into multiple scriptlets, you can put HTML or other directives in the middle, and those pieces are conditionally executed when the stored procedure is run.

### **Syntax**

```
<% PL/SQL statement;  
    [ PL/SQL statement; ] ... %>
```

## Expression Block

Specifies a single PL/SQL expression, such as a string, arithmetic expression, function call, or combination of those things. The result is substituted as a string at that spot in the HTML page that is produced by the stored procedure. You do not need to end the PL/SQL expression with a semicolon.

### Syntax

```
<%= PL/SQL expression %>
```

## Loading the PL/SQL Server Page into the Database as a Stored Procedure

You load one or more PSP files into the database as stored procedures. Each .psp file corresponds to one stored procedure. The pages are compiled and loaded in one step, to speed up the development cycle:

```
loadpsp [ -replace ] -user username/password[@connect_string]  
        [ include_file_name ... ] psp_file_name ...
```

To do a "create and replace" on the stored procedures, include the -replace flag.

The loader logs on to the database using the specified user name, password, and connect string. The stored procedures are created in the corresponding schema.

Include the names of all the include files (whose names do not have the .psp extension) before the names of the PL/SQL server pages (whose names have the .psp extension).

For example:

```
loadpsp -replace -user scott/tiger@WEBDB banner.inc display_order.psp
```

In this example:

- The stored procedure is created in the database WEBDB. The database is accessed as user scott with password tiger, both to create the stored procedure and when the stored procedure is executed.
- timestamp.inc and banner.inc are files containing boilerplate text and script code, that are included by the .psp file.
- display\_order.psp contains the main code and text for the web page. By default, the corresponding stored procedure is named DISPLAY\_ORDER.

## Running a PL/SQL Server Page via a URL

Once the PL/SQL server page has been turned into a stored procedure, you can run it by retrieving an HTTP URL through a web browser or other Internet-aware client program. The virtual path in the URL depends on the way the web gateway is configured.

The parameters to the stored procedure are passed through the CGI mechanism, either the POST method or the GET method. With the POST method, the parameters are passed directly from an HTML form and are not visible in the URL. With the GET method, the parameters are passed in the query string of the URL, separated by & characters, with most non-alphanumeric characters in encoded format (such as %20 for a space). You can use the GET method to call a PSP page from an HTML form, or you can use a hardcoded HTML link to call the stored procedure with a given set of parameters.

### Sample PSP URLs

Using METHOD=GET, the URL might look something like this:

```
http://sitename/schemaname/pspname?parmname1=value1&parmname2=value2
```

Using METHOD=POST, the URL does not show the parameters:

```
http://sitename/schemaname/pspname
```

The METHOD=GET format is more convenient for debugging and allows visitors to pass exactly the same parameters when they return to the page through a bookmark.

The METHOD=POST format allows a larger volume of parameter data, and is suitable for passing sensitive information that should not be displayed in the URL. (URLs linger on in the browser's history list and in the CGI variables that are passed to the next-visited page.) It is not practical to bookmark pages that are called this way.

## Examples of PL/SQL Server Pages

This section shows how you might start with a very simple PL/SQL server page, and produce progressively more complicated versions as you gain more confidence.

As you go through each step, you can use the procedures in "[Loading the PL/SQL Server Page into the Database as a Stored Procedure](#)" on page 15-11 and "[Running a PL/SQL Server Page via a URL](#)" on page 15-12 to compile the PSP files and try them in a browser.

## Sample Table

In this example, we use a very small table representing a product catalog. It holds the name of an item, the price, and URLs for a description and picture of the item.

Name	Type
PRODUCT	VARCHAR2(100)
PRICE	NUMBER(7,2)
URL	VARCHAR2(200)
PICTURE	VARCHAR2(200)

```
Guitar
455.5
http://auction.fictional_site.com/guitar.htm
http://auction.fictional_site.com/guitar.jpg
```

```
Brown shoe
79.95
http://retail.fictional_site.com/loafers.htm
http://retail.fictional_site.com/shoe.gif
```

```
Radio
9.95
http://promo.fictional_site.com/freegift.htm
http://promo.fictional_site.com/alarmclock.jpg
```

## Dumping the Sample Table

For your own debugging, you might want to display the complete contents of an SQL table. You can do this with a single call to `OWA_UTIL.TABLEPRINT`. In subsequent iterations, we use other techniques to get more control over the presentation.

```
<@% page language="PL/SQL" %>
<@% plsql procedure="show_catalog_simple" %>
<HTML>
<HEAD><TITLE>Show Contents of Catalog (Complete Dump)</TITLE></HEAD>
<BODY>
<%
declare
dummy boolean;
begin
dummy := owa_util.tableprint('catalog','border');
```

```
end;  
%>  
</BODY>  
</HTML>
```

## Printing the Sample Table using a Loop

Next, we loop through the items in the table and explicitly print just the pieces we want.

- We could adjust the SELECT statement to retrieve only a subset of the rows or columns.
- We could change the HTML or the location of the expressions to change the appearance of each item, or the order in which the columns are shown.
- At this early stage, we pick a very simple presentation, a set of list items, to avoid any problems from mismatched or unclosed table tags.

```
<@% page language="PL/SQL" %>  
<@% plsql procedure="show_catalog_raw" %>  
<HTML>  
<HEAD><TITLE>Show Contents of Catalog (Raw Form)</TITLE></HEAD>  
<BODY>  
<UL>  
<% for item in (select * from catalog order by price desc) loop %>  
<LI>  
Item = <%= item.product %><BR>  
price = <%= item.price %><BR>  
URL = <I><%= item.url %></I><BR>  
picture = <I><%= item.picture %></I>  
<% end loop; %>  
</UL>  
</BODY>  
</HTML>
```

Once the previous simple example is working, we can display the contents in a more usable format.

- We use some HTML tags around certain values for emphasis.
- Instead of printing the URLs for the description and picture, we plug them into link and image tags so that the reader can see the picture and follow the link.

```
<@% page language="PL/SQL" %>
```

```

<%@ plsql procedure="show_catalog_pretty" %>
<HTML>
<HEAD><TITLE>Show Contents of Catalog (Better Form)</TITLE></HEAD>
<BODY>
<UL>
<% for item in (select * from catalog order by price desc) loop %>
<LI>
Item = <A HREF="<%= item.url %>"><%= item.product %></A><BR>
price = <BIG><%= item.price %></BIG><BR>
<IMG SRC="<%= item.picture %>">
<% end loop; %>
</UL>
</BODY>
</HTML>

```

## Allowing a User Selection

Now we have a dynamic page, but from the user's point of view it may still be dull. The results are always the same unless you update the catalog table.

- To liven up the page, we can make it accept a minimum price, and present only the items that are more expensive. (Your customers' buying criteria may vary.)
- When the page is displayed in a browser, by default the minimum price is 100 units of the appropriate currency. Later, we will see how to allow the user to pick a minimum price.

```

<%@ page language="PL/SQL" %>
<%@ plsql procedure="show_catalog_partial" %>
<%@ plsql parameter="minprice" default="100" %>

<HTML>
<HEAD><TITLE>Show Items Greater Than Specified Price</TITLE></HEAD>
<BODY>
<P>This report shows the items whose price is greater than <%= minprice %>.
<UL>
<% for item in (select * from catalog where price > minprice order by price
desc) loop %>
<LI>
Item = <A HREF="<%= item.url %>"><%= item.product %></A><BR>
price = <BIG><%= item.price %></BIG><BR>
<IMG SRC="<%= item.picture %>">
<% end loop; %>
</UL>

```

```
</BODY>
</HTML>
```

The above technique of filtering results is fine for some applications, such as search results, where users might worry about being overwhelmed by choices. But in a retail situation, you might want to use an alternative technique so that customers can still choose to purchase other items.

- Instead of filtering the results through a WHERE clause, we can retrieve the entire result set, then take different actions for different returned rows.
- We can change the HTML to highlight the output that meets their criteria. In this case, we use the background color for an HTML table row. We could also insert a special icon, increase the font size, or use some other technique to call attention to the most important rows.
- At this point, where we want to present a specific user experience, it becomes worth the trouble to lay out the results in an HTML table.

```
<@@ page language="PL/SQL" %>
<@@ plsql procedure="show_catalog_highlighted" %>
<@@ plsql parameter="minprice" default="100" %>
<%! color varchar2(7); %>

<HTML>
<HEAD><TITLE>Show Items Greater Than Specified Price</TITLE></HEAD>
<BODY>
<P>This report shows all items, highlighting those whose price is
  greater than <%= minprice %>.
<TABLE BORDER>
<TR>
<TH>Product</TH>
<TH>Price</TH>
<TH>Picture</TH>
</TR>
<%
for item in (select * from catalog order by price desc) loop
  if item.price > minprice then
    color := '#CCCCFF';
  else
    color := '#CCCCCC';
  end if;
%>
<TR BGCOLOR="<%= color %>">
<TD><A HREF="<%= item.url %>"><%= item.product %></A></TD>
<TD><BIG><%= item.price %></BIG></TD>
```

```
<TD><IMG SRC="<%= item.picture %>"></TD>
</TR>
<% end loop; %>
</TABLE>
</BODY>
</HTML>
```

### Sample HTML Form to Call a PL/SQL Server Page

Here is a bare-bones HTML form that allows someone to enter a price, and then calls the `SHOW_CATALOG_PARTIAL` stored procedure passing the entered value as the `MINPRICE` parameter.

To avoid coding the entire URL of the stored procedure in the `ACTION=` attribute of the form, we can make the form a PSP file so that it goes in the same directory as the PSP file it calls. Even though this HTML file has no PL/SQL code, we can give it a `.psp` extension and load it as a stored procedure into the database. When the stored procedure is run, it just displays the HTML exactly as it appears in the file.

```
<html>
<body>
<form method="POST" action="show_catalog_partial">
<p>Enter the minimum price you want to pay:
<input type="text" name="minprice">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

---

---

**Note:** An HTML form is different from other forms you might produce with tools and programming languages. It is part of an HTML file, delimited by `<FORM>` and `</FORM>` tags, where someone can make choices and enter data, then transmit those choices to a server-side program using the CGI protocol.

To produce a complete application using PSP, you might need to learn the syntax of `<INPUT>`, `<SELECT>`, and other HTML tags related to forms.

---

---

## Debugging PL/SQL Server Page Problems

As you begin experimenting with PSP, and as you adapt your first simple pages into more elaborate ones, keep these guidelines in mind when you encounter problems:

- The first step is to get all the PL/SQL syntax and PSP directive syntax right. If you make a mistake here, the file does not compile.
  - Make sure you use semicolons to terminate lines where required.
  - If a value must be quoted, quote it. You might need to enclose a single-quoted value (needed by PL/SQL) inside double quotes (needed by PSP).
  - Mistakes in the PSP directives are usually reported through PL/SQL syntax messages. Check that your directives use the right syntax, that directives are closed properly, and that you are using the right element (declaration, expression, or code block) depending on what goes inside it.
  - PSP attribute names are case-sensitive. Most are specified in all lowercase; `contentType` and `errorPage` must be specified as mixed-case.
- The next step is to run the PSP file by requesting its URL in a web browser. At this point, you might get an error that the file is not found.
  - Make sure you are requesting the right virtual path, depending on the way the web gateway is configured. Typically, the path includes the hostname, optionally a port number, the schema name, and the name of the stored procedure (with no `.psp` extension).
  - Remember, if you use the `-replace` option when compiling the file, the old version of the stored procedure is erased. So, after a failed compilation, you must fix the error or the page is not available. You might want to test new scripts in a separate schema until they are ready, then load them into the production schema.

- If you copied the file from another file, remember to change any procedure name directives in the source to match the new file name.
- Once you get one file-not-found error, make sure to request the latest version of the page the next time. The error page might be cached by the browser. You might need to press Shift-Reload in the browser to bypass its cache.
- When the PSP script is run, and the results come back to the browser, use standard debugging techniques to check for and correct wrong output. The tricky part is to set up the interface between different HTML forms, scripts, and CGI programs so that all the right values are passed into your page. The page might return an error because of a parameter mismatch.
  - To see exactly what is being passed to your page, use METHOD=GET in the calling form so that the parameters are visible in the URL.
  - Make sure that the form or CGI program that calls your page passes the correct number of parameters, and that the names specified by the NAME= attributes on the form match the parameter names in the PSP file. If the form includes any hidden input fields, or uses the NAME= attribute on the Submit or Reset buttons, the PSP file must declare equivalent parameters.
  - Make sure that the parameters can be cast from string into the correct PL/SQL types. For example, do not include alphabetic characters if the parameter in the PSP file is declared as a NUMBER.
  - Make sure that the URL's query string consists of name-value pairs, separated by equals signs, especially if you are passing parameters by constructing a hardcoded link to the page.
  - If you are passing a lot of parameter data, such as large strings, you might exceed the volume that can be passed with METHOD=GET. You can switch to METHOD=POST in the calling form without changing your PSP file.

## Putting an Application using PL/SQL Server Pages into Production

When you start developing an application with PSP, you may spend most of your time getting the logic correct in the script. Before putting the application into production, consider other issues such as usability and download speed:

- Pages can be rendered faster in the browser if the HEIGHT= and WIDTH= attributes are specified for all images. You might standardize on picture sizes, or store the height and width of images in the database along with the data or URL.

- For viewers who turn off graphics, or who use alternative browsers that read the text out loud, include a description of significant images using the ALT= attribute. That is another string you might store in the database along with the image.
- Although an HTML table provides a good way to display data, a large table can make your application seem slow; often, the viewer sees a blank page until the entire table is downloaded. If the amount of data in an HTML table is large, consider splitting the output into multiple tables.
- If you set text, font, or background colors, test your application with different combinations of browser color settings:
  - Test what happens if you override just the foreground color in the browser, or just the background color, or both.
  - Generally, if you set one color (such as the foreground text color), you should set all the colors through the <BODY> tag, to avoid hard-to-read combinations like white text on a white background.
  - If you use a background image, specify a similar background color to provide proper contrast for viewers who do not load graphics.
  - If the information conveyed by different colors is crucial, consider using some other method instead of or in addition to the color change. For example, you might put a graphic icon next to special items in a table. Some of your viewers may see your page on a monochrome screen, or on browsers that cannot represent different colors. (Such browsers might fit in a shirt pocket and use a stylus for input.)
- Providing context information prevents users from getting lost. Include a descriptive <TITLE> tag for your page. If the user is partway through a procedure, indicate which step is represented by your page. Provide links to logical points to continue with the procedure, return to a previous step, or cancel the procedure completely. Many pages might use a standard set of links that you embed using the include directive.
- In any entry fields, users might enter incorrect values. Where possible, use select lists to present a set of choices. Validate any text entered in a field before passing it to SQL. The earlier you can validate, the better; a Javascript routine can detect incorrect data and prompt the user to correct it before they press the Submit button and make a call to the database.
- Browsers tend to be lenient when displaying incorrect HTML. But what looks OK in one browser might look bad or might not display at all in another browser.

- Pay attention to HTML rules for quotation marks, closing tags, and especially for anything to do with tables.
- Minimize the dependence on tags that are only supported by a single browser. Sometimes you can provide an extra bonus using such tags, but your application should still be usable with other browsers.
- You can check the validity, and even in some cases the usability, of your HTML for free at many sites on the World Wide Web.



---

# Working with Transaction Monitors with Oracle XA

This chapter describes how to use the Oracle XA library, which is typically used in applications that work with transaction monitors. The XA features are most useful in applications where transactions interact with more than one database.

The Oracle XA library is an external interface that allows global transactions to be coordinated by a transaction manager other than the Oracle8 Server. This allows inclusion of non-Oracle8 Server entities called resource managers (RM) in distributed transactions.

The Oracle XA library conforms to the X/Open Distributed Transaction Processing (DTP) software architecture's XA interface specification.

The chapter includes the following topics:

- [X/Open Distributed Transaction Processing \(DTP\)](#)
- [XA and the Two-Phase Commit Protocol](#)
- [Transaction Processing Monitors \(TPMs\)](#)
- [Support for Dynamic and Static Registration](#)
- [Oracle XA Library Interface Subroutines](#)
- [Developing and Installing Applications That Use the XA Libraries](#)
- [Troubleshooting XA Applications](#)
- [General XA Issues and Restrictions](#)
- [Changes to Oracle XA Support](#)

**See Also:**

- For a general overview of XA, including basic architecture, see *X/Open CAE Specification - Distributed Transaction Processing: The XA Specification*. You can obtain a copy of this document by requesting X/Open Document No. XO/CAE/91/300 or ISBN 1 872630 24 3 from:  
  
X/Open Company, Ltd., 1010 El Camino Real, Suite 380, Menlo Park, CA 94025, U.S.A.
- For background and reference information about the Oracle XA library, see *Oracle Call Interface Programmer's Guide*.
- For information on library linking filenames, see the Oracle operating system-specific documentation.
- A `README.doc` file is located in a directory specified in the Oracle operating system-specific documentation and describes changes, bugs, or restrictions in the Oracle XA library for your platform since the last version.

## X/Open Distributed Transaction Processing (DTP)

The X/Open DTP architecture defines a standard architecture or interface that allows multiple application programs to share resources, provided by multiple, and possibly different, resource managers. It coordinates the work between application programs and resource managers into global transactions.

[Figure 16-1](#) illustrates a possible X/Open DTP model.

A resource manager (RM) controls a shared, recoverable resource that can be returned to a consistent state after a failure. For example, Oracle8 Server is an RM and uses its redo log and undo segments to return to a consistent state after a failure. An RM provides access to shared resources such as a database, file systems, printer servers, and so forth.

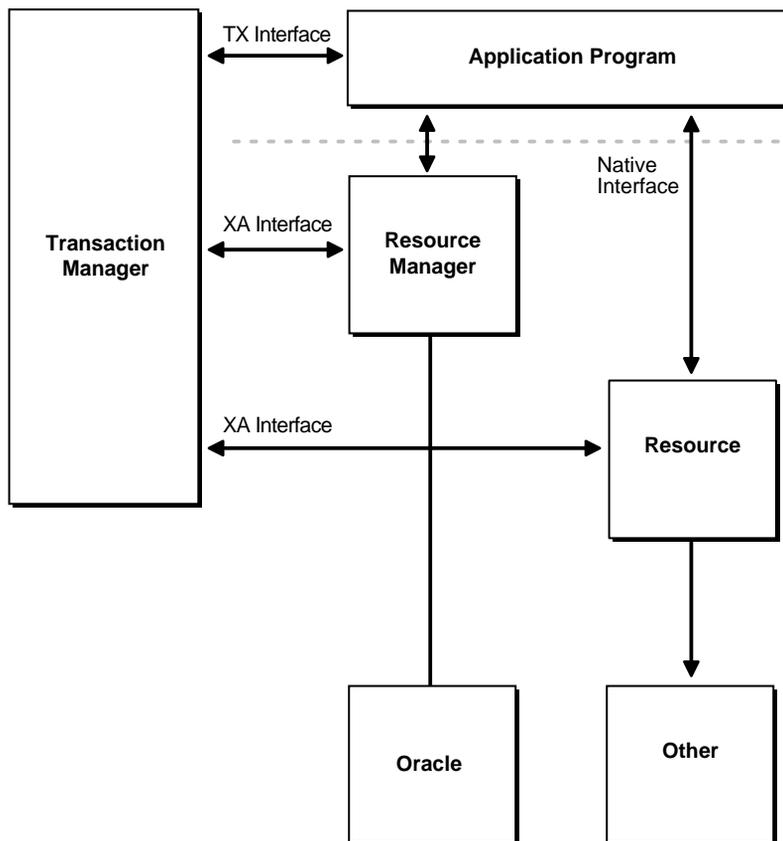
A transaction manager (TM) provides an application program interface (API) for specifying the boundaries of the transaction and manages the commit and recovery procedures.

Normally, Oracle8 Server acts as its own TM and manages its own commit and recovery. However, using a standards-based TM allows Oracle8 Server to cooperate with other heterogeneous RMs in a single transaction.

A TM is usually a component provided by a transaction processing monitor (TPM) vendor. The TM assigns identifiers to transactions, and monitors and coordinates their progress. It uses Oracle XA library subroutines to tell Oracle8 Server how to process the transaction, based on its knowledge of all RMs in the transaction. You can find a list of the XA subroutines and their descriptions later in this section.

An application program (AP) defines transaction boundaries and specifies actions that constitute a transaction. For example, an AP can be a precompiler or OCI program. The AP operates on the RM's resource through the RM's native interface, for example SQL. However, it starts and completes all transaction operations via the transaction manager through an interface called TX. The AP itself does not directly use the XA interface

**Figure 16-1 One Possible DTP Model**



---



---

**Note:** The naming conventions for the TX interface and associated subroutines are vendor-specific, and may differ from those used here. For example, you may find that the `tx_open` call is referred to as `tp_open` on your system. To check terminology, see the documentation supplied with the transaction processing monitor.

---



---

## Required Public Information

As a resource manager, Oracle is required to publish the following information.

<code>xa_switch_t</code> structures	The Oracle Server <code>xa_switch_t</code> structure name for static registration is <code>xaosw</code> . The Oracle Server <code>xa_switch_t</code> structure name for dynamic registration is <code>xaoswd</code> . These structures contain entry points and other information for the resource manager.
<code>xa_switch_t</code> resource manager	The Oracle Server resource manager name within the <code>xa_switch_t</code> structure is <code>Oracle_XA</code> .
close string	The close string used by <code>xa_close ()</code> is ignored and is allowed to be null.
open string	The format of the open string used by <code>xa_open ()</code> is described in detail in " <a href="#">Defining the xa_open String</a> " on page 16-10.
libraries	Libraries needed to link applications using Oracle XA have operating system-specific names. It is similar to linking an ordinary precompiler or OCI program except you may have to link any TPM-specific libraries. If you are not using <code>sqllib</code> , then be sure to link with <code>\$ORACLE_HOME/lib/xaonsl.o</code> .
requirements	A purchased and installed distributed database option.

## XA and the Two-Phase Commit Protocol

The Oracle XA library interface follows the two-phase commit protocol, consisting of a prepare phase and a commit phase, to commit transactions.

In phase one, the prepare phase, the TM asks each RM to guarantee the ability to commit any part of the transaction. If this is possible, then the RM records its prepared state and replies affirmatively to the TM. If it is not possible, then the RM

may roll back any work, reply negatively to the TM, and forget any knowledge about the transaction. The protocol allows the application, or any RM, to roll back the transaction unilaterally until the prepare phase is complete.

In phase two, the commit phase, the TM records the commit decision. Then the TM issues a commit or rollback to all RMs which are participating in the transaction.

---

---

**Note:** TM can issue a commit for an RM only if all RMs have replied affirmatively to phase one.

---

---

## Transaction Processing Monitors (TPMs)

A transaction processing monitor (TPM) coordinates the flow of transaction requests between the client processes that issue requests and the back-end servers that process them. Basically, it coordinates transactions that require the services of several different types of back-end processes, such as application servers and resource managers that are distributed over a network.

The TPM synchronizes any commits and rollbacks required to complete a distributed transaction. The transaction manager (TM) portion of the TPM is responsible for controlling when distributed commits and rollbacks take place. Thus, if a distributed application program is written to take advantage of a TPM, then the TM portion of the TPM is responsible for controlling the two-phase commit protocol. The RMs enable the TMs to do this.

Because the TM controls distributed commits or rollbacks, it must communicate directly with Oracle (or any other resource manager) through the Oracle XA library interface.

## Support for Dynamic and Static Registration

The Oracle8 server supports both dynamic and static registration. In dynamic registration, the RM executes an application callback before starting any work. In static registration, you must call `xa_start` for each RM before starting any work, even if some RMs are not involved.

To use dynamic registration, both client and server must be at Oracle 8.0 or later. Otherwise, you can only use static registration.

## Oracle XA Library Interface Subroutines

The Oracle XA library subroutines allow a TM to instruct an Oracle8 Server what to do about transactions. Generally, the TM must "open" the resource (using `xa_open`). Typically, this results from the AP's call to `tx_open`. Some TMs may call `xa_open` implicitly, when the application begins. Similarly, there is a close (using `xa_close`) that occurs when the application is finished with the resource. This may be when the AP calls `tx_close` or when the application terminates.

There are several other tasks the TM instructs the RMs to do. These include among others:

- Starting a new transaction and associating it with an ID
- Rolling back a transaction
- Preparing and committing a transaction

### XA Library Subroutines

The following XA Library subroutines are available:

<code>xa_open</code>	Connects to the resource manager.
<code>xa_close</code>	Disconnects from the resource manager.
<code>xa_start</code>	Starts a new transaction and associate it with the given transaction ID (XID), or associates the process with an existing transaction.
<code>xa_end</code>	Disassociates the process from the given XID.
<code>xa_rollback</code>	Rolls back the transaction associated with the given XID.
<code>xa_prepare</code>	Prepares the transaction associated with the given XID. This is the first phase of the two-phase commit protocol.
<code>xa_commit</code>	Commits the transaction associated with the given XID. This is the second phase of the two-phase commit protocol.
<code>xa_recover</code>	Retrieves a list of prepared, heuristically committed or heuristically rolled back transaction.
<code>xa_forget</code>	Forgets the heuristic transaction associated with the given XID.

In general, the AP does not need to worry about these subroutines except to understand the role played by the `xa_open` string.

## Extensions to the XA Interface

Oracle's XA interface includes some additional functions:

1. `OCIsvcCtx *xaoSvcCtx(text *dbname):`

This function returns the OCI service handle for a given XA connection. The `dbname` parameter must be the same as the `dbname` parameter passed in the `xa_open` string. OCI applications can use this routing instead of the `sqlld2` calls to obtain the connection handle. Hence, OCI applications need not link with the `SQLLIB` library. The service handle can be converted to the Version 7 OCI logon data area (LDA) using `OCIsvcCtxToLda()` [Version 8 OCI]. Client applications must remember to convert the Version 7 LDA to a service handle using `OCILdaToSvcCtx()` after completing the OCI calls.

2. `OCIEnv *xaoEnv(text *dbname):`

This function returns the OCI environment handle for a given XA connection. The `dbname` parameter must be the same as the `dbname` parameter passed in the `xa_open` string.

3. `int xaosterr(OCIsvcCtx *SvcCtx, sb4 error):`

This function, only applicable to dynamic registration, converts an Oracle error code to an XA error code. The first parameter is the service handle used to execute the work in the database. The second parameter is the error code that was returned from Oracle. Use this function to determine if the error returned from an OCI command was caused because the `xa_start` failed. The function returns `XA_OK` if the error was not generated by the XA module and a valid XA error if the error was generated by the XA module.

## Developing and Installing Applications That Use the XA Libraries

This section discusses developing and installing Oracle8 Server applications:

- [Responsibilities of the DBA or System Administrator](#) on page 16-9
- [Responsibilities of the Application Developer](#) on page 16-10
- [Defining the xa\\_open String](#) on page 16-10
- [Interfacing to Precompilers and OCIs](#) on page 16-17
- [Transaction Control using XA](#) on page 16-20
- [Migrating Precompiler or OCI Applications to TPM Applications](#) on page 16-23
- [XA Library Thread Safety](#) on page 16-24

### Responsibilities of the DBA or System Administrator

The responsibilities of the DBA or system administrator are

1. Define the open string with the application developer's help.  
This is described in "[Defining the xa\\_open String](#)" on page 16-10.
2. Make sure the DBA\_PENDING\_TRANSACTIONS view exists on the database.

#### For Oracle Server Release 8.0:

Grant the select privilege to the DBA\_PENDING\_TRANSACTIONS view for all Oracle Server *user(s)* specified in the `xa_open` string.

#### For Oracle Server Release 7.3:

Make sure V\$XATRANS\$ exists.

This view should have been created during the XA library installation. You can manually create the view, if needed, by running the SQL script XAVIEW.SQL. This SQL script should be executed as the Oracle user SYS. Grant the SELECT privilege to the V\$XATRANS\$ view for all Oracle Server accounts which will be used by Oracle XA library applications.

**See Also:** Your Oracle operating system-specific documentation contains the location of the XAVIEW.SQL script.

3. Install the resource manager, using the open string information, into the TPM configuration, following the TPM vendor instructions.

The DBA or system administrator should be aware that a TPM system starts the process that connects to an Oracle8 Server. See your TPM documentation to determine what environment exists for the process and what user ID it will have.

Be sure that correct values are set for `ORACLE_HOME` and `ORACLE_SID`.

**See Also:** ["Defining the xa\\_open String"](#) on page 16-10 has information on how to specify a *sid* or a trace directory that is different from the defaults.

Also be sure to grant the user the `SELECT` privilege on `DBA_PENDING_TRANSACTIONS`.

4. Start up the relevant databases to bring Oracle XA applications on-line.

This should be done before starting any TPM servers.

## Responsibilities of the Application Developer

The application developer's responsibilities are

1. Define the open string with the DBA or system administrator's help.

Defining the open string is described later in this section.

2. Develop the applications.

Observe special restrictions on transaction-oriented SQL statements for precompilers.

**See Also:** ["Interfacing to Precompilers and OCIs"](#) on page 16-17

3. Link the application according to TPM vendor instructions.

## Defining the xa\_open String

The open string is used by the transaction monitor to open the database. The maximum number of characters in an open string is 256.

This section covers:

- [Syntax of the xa\\_open String](#)
- [Required Fields](#)
- [Optional Fields](#)

## Syntax of the xa\_open String

Oracle\_XA{+required\_fields...} [+optional\_fields...]

where *required\_fields* are:

Acc=P//

Or

Acc=P/user/password

SesTm=session\_time\_limit

and where *optional\_fields* are:

DB=db\_name

LogDir=log\_dir

MaxCur=maximum\_#\_of\_open\_cursors

Objects=true/false

SqlNet=connect\_string

Loose\_Coupling=true/false

SesWt=session\_wait\_limit

Threads=true/false

---



---

### Note:

- You can enter the required fields and optional fields *in any order* when constructing the open string.
  - All field names are case insensitive. Their values may or may not be case-sensitive depending on the platform.
  - There is no way to use the "+" character as part of the actual information string.
- 
- 

## Required Fields

Required fields for the open string are described in this section.

Acc=P//

or

*Acc=P/user/password*

<i>Acc</i>	Specifies user access information
<i>P</i>	Indicates that explicit user and password information is provided.
<i>P//</i>	Indicates that no explicit user or password information is provided, and that the operating system authentication form will be used.  For more information see <i>Oracle8i Administrator's Guide</i> .
<i>user</i>	A valid Oracle Server account.
<i>password</i>	The corresponding current password.

For example, *Acc=P/scott/tiger* indicates that user and password information is provided. In this case, the user is *scott* and the password is *tiger*.

As previously mentioned, make sure that *scott* has the `SELECT` privilege on the `DBA_PENDING_TRANSACTIONS` table.

*Acc=P//* indicates that no user or password information is provided, thus defaulting to operating system authentication.

*SesTm=session\_time\_limit*

<i>SesTm</i>	Specifies the maximum length of time a transaction can be inactive before it is automatically aborted by the system.
--------------	--

***session\_time\_limit***

This value should be the maximum time allowed in a transaction between one service and the next, or a service and the commit or rollback of the transaction.

For example, if the TPM uses remote procedure calls between the client and the servers, then *SesTM* applies to the time between the completion of one RPC and the initiation of the next RPC, or the *tx\_commit*, or the *tx\_rollback*.

The unit for this time limit is in seconds. The value of 0 indicates no limit. For example, *SesTM=15* indicates that the session idle time limit is 15 seconds.

Entering a value of 0 is strongly discouraged. It might tie up resources for a long time if something goes wrong. Also, if a child process has *SesTM=0*, the *SesTM* setting is not effective after the parent process is terminated.

**Optional Fields**

Optional fields are described below.

*DB=db\_name*

DB Specifies the database name.

*db\_name* Indicates the name used by Oracle precompilers to identify the database.

Application programs that use only the default database for the Oracle precompiler (that is, they do not use the AT clause in their SQL statements) should omit the `DB=db_name` clause in the open string.

Applications that use explicitly named databases should indicate that database name in their `DB=db_name` field.

Version 7 OCI programs need to call the `sqlld20` function to obtain the correct `lda_def`, which is the equivalent of a service context. Version 8 OCI programs need to call the `xaoSvcCtx` function to get the `OCISvcCtx` service context.

The *db\_name* is not the *sid* and is not used to locate the database to be opened. Rather, it correlates the database opened by this open string with the name used in the application program to execute SQL statements. The *sid* is set from either the environment variable `ORACLE_SID` of the TPM application server or the *sid* given in the Net8 (formerly, SQL\*Net) clause in the open string. The Net8 clause is described later in this section.

Some TPM vendors provide a way to name a group of servers that use the same open string. The DBA may find it convenient to choose the same name both for that purpose and for *db\_name*.

For example, `DB=payroll` indicates that the database name is "payroll", and that the application server program will use that name in AT clauses.

`LogDir=log_dir`

**LogDir** Specifies the directory on a local machine where the Oracle XA library error and tracing information may be logged.

*log\_dir* Indicates the pathname of the directory where the tracing information should be stored. The default is `$ORACLE_HOME/rdbms/log` if `ORACLE_HOME` is set; otherwise, it is the current directory.

For example, `LogDir=/xa_trace` indicates that the error and tracing information is located under the `/xa_trace` directory.

---



---

**Note:** Ensure that the directory you specify for logging exists and the application server can write to it.

---



---

`Loose_Coupling=true/false`

See "[Transaction Branches](#)" on page 16-31 for a complete explanation.

`Objects=true/false`

<b>Objects</b>	Specifies whether the application process is initialized in object mode. The default value is <code>False</code> .
<i>true/false</i>	If the application needs to use certain API calls that require object mode, such as <code>OCIAssignRawbytes()</code> , then set the value to <i>true</i> .

`MaxCur=maximum_#_of_open_cursors`

<b>MaxCur</b>	Specifies the number of cursors to be allocated when the database is opened. It serves the same purpose as the precompiler option <code>maxopencursors</code> .
---------------	---

<i>maximum_#_of_</i> <i>open_cursors</i>	Indicates the number of open cursors to be cached.
---	--

For example, `MaxCur=5` indicates that the precompiler should try to keep five open cursors cached.

---



---

**Note:** This parameter overrides the precompiler option `maxopencursors` that you might have specified in your source code or at compile time.

---



---

**See Also:** *Pro\*C/C++ Precompiler Programmer's Guide*

`SqlNet=db_link`

<code>SqlNet</code>	Specifies the Net8 (formerly, SQL*Net) database link.
<code>db_link</code>	Indicates the string to use to log on to the system. The syntax for this string is the same as that used to set the <code>TWO-TASK</code> environment variable.

For example, `SqlNet=hqfin@NEWDB` indicates the database with `sid=NEWDB` accessed at host `hqfin` by TCP/IP.

The `SqlNet` parameter can be used to specify the `ORACLE_SID` in cases where you cannot control the server environment variable. It must also be used when the server needs to access more than one Oracle Server database. To use the Net8 string without actually accessing a remote database, use the Pipe driver.

For example:

```
SqlNet=localsid1
```

Where:

`localsid1`            An alias defined in the Net8 `tnsnames.ora` file.

Make sure that all databases to be accessed with a Net8 database link have an entry in `/etc/oratab`.

```
SesWt=session_wait_limit
```

`SesWt`                Specifies the time-out limit when waiting for a transaction branch that is being used by another session. The default value is 60 seconds.

*session\_wait\_limit*    The number of seconds Oracle waits before `XA_RETRY` is returned.

```
Threads=true/false
```

`Threads`             Specifies whether the application is multi-threaded. The default value is `False`.

*true/false*            If the application is multi-threaded, then the setting is *true*.

## Interfacing to Precompilers and OCIs

This section describes how to use the Oracle XA library with precompilers and Oracle Call Interfaces (OCIs).

### Using Precompilers with the Oracle XA Library

When used in an Oracle XA application, cursors are valid only for the duration of the transaction. Explicit cursors should be opened after the transaction begins, and closed before the commit or rollback.

There are two options to choose from when interfacing with precompilers:

- Using precompilers with the default database
- Using precompilers with a named database

The following examples use the precompiler Pro\*C/C++.

### Using Precompilers with the Default Database

To interface to a precompiler with the default database, make certain that the `DB=db_name` field, used in the open string, is not present. The absence of this field indicates the default connection, and only one default connection is allowed per process.

The following is an example of an open string identifying a default Pro\*C/C++ connection.

```
ORACLE_XA+SqlNet=host@MAIL+ACC=P/scott/tiger
+SesTM=10+LogDir=/usr/local/logs
```

Note that the `DB=db_name` is absent, indicating an empty database ID string.

The syntax of a SQL statement would be:

```
EXEC SQL UPDATE Emp_tab SET Sal = Sal*1.5;
```

### Using Precompilers with a Named Database

To interface to a precompiler with a named database, include the `DB=db_name` field in the open string. Any database you refer to must reference the same `db_name` you specified in the corresponding open string.

An application may include the default database, as well as one or more named databases, as shown in the following examples.

For example, suppose you want to update an employee's salary in one database, his department number (DEPTNO) in another, and his manager in a third database. You would configure the following open strings in the transaction manager:

```
ORACLE_XA+DB=MANAGERS+SqlNet=hqfin@SID1+ACC=P/scott/tiger
+SesTM=10+LogDir=/usr/local/xalog
ORACLE_XA+DB=PAYROLL+SqlNet=SID2+ACC=P/scott/tiger
+SesTM=10+LogDir=/usr/local/xalog
ORACLE_XA+SqlNet=hqemp@SID3+ACC=P/scott/tiger
+SesTM=10+LogDir=/usr/local/xalog
```

Note that there is no `DB=db_name` field in the last open string.

In the application server program, you would enter declarations, such as:

```
EXEC SQL DECLARE PAYROLL DATABASE;
EXEC SQL DECLARE MANAGERS DATABASE;
```

Again, the default connection (corresponding to the third open string that does not contain the `db_name` field) needs no declaration.

When doing the update, you would enter statements similar to the following:

```
EXEC SQL AT PAYROLL UPDATE Emp_Tab SET Sal=4500 WHERE Empno=7788;
EXEC SQL AT MANAGERS UPDATE Emp_Tab SET Mgr=7566 WHERE Empno=7788;
EXEC SQL UPDATE Emp_Tab SET Deptno=30 WHERE Empno=7788;
```

There is no `AT` clause in the last statement because it is referring to the default database.

In Oracle precompilers release 1.5.3 or later, you can use a character host variable in the `AT` clause, as the following example shows:

```
EXEC SQL BEGIN DECLARE SECTION;
    DB_NAME1 CHARACTER(10);
    DB_NAME2 CHARACTER(10);
EXEC SQL END DECLARE SECTION;

.
.
SET DB_NAME1 = 'PAYROLL'
SET DB_NAME2 = 'MANAGERS'

.
.
EXEC SQL AT :DB_NAME1 UPDATE...
EXEC SQL AT :DB_NAME2 UPDATE...
```

---



---

**Caution: Oracle recommends against using XA applications to create connections. Any work performed would be outside the global transaction and would have to be committed separately.**

---



---

## Using OCI with the Oracle XA Library

OCI applications that use the Oracle XA library should not call `OCISessionBegin()` (`olon()` or `orlon()` in Version 7) to log on to the resource manager. Rather, the logon should be done through the TPM. The applications can execute the function `xaoSvcCtx()` (`sqlld2()` in Version 7) to obtain the service context (`lda` in Version 7) structure they need to access the resource manager.

In applications that need to pass the environment handle to OCI functions, you can also call `xaoEnv()` to find that handle.

Because an application server can have multiple concurrent open Oracle Server resource managers, it should call the function `xaoSvcCtx()` with the correct arguments to obtain the correct service context.

### Release 7.3

If `DB=db_name` is not present in the open string, then execute:

```
sqlld2(lda, NULL, 0);
```

This obtains the `lda` for this resource manager.

Alternatively, if `DB=db_name` is present in the open string, then execute:

```
sqlld2(lda, db_name, strlen(db_name));
```

This obtains the `lda` for this resource manager.

### Release 8.0

If `DB=db_name` is not present in the open string, then execute:

```
xaoSvcCtx(NULL);
```

Alternatively, if `DB=db_name` is present in the open string, then execute:

```
xaoSvcCtx(db_name);
```

This gets the server context for this resource manager.

In the same way, you can execute:

```
xaoEnv(NULL);
```

or:

```
xaoEnv(db_name);
```

depending upon the open string, to get the environment handle.

**See Also:** *Oracle Call Interface Programmer's Guide* has more information about using the `OCISvcCtx`.

## Transaction Control using XA

This section explains how to use transaction control within the Oracle XA library environment.

When the XA library is used, transactions are not controlled by the SQL statements that commit or roll back transactions. Rather, they are controlled by an API accepted by the TM that starts and stops transactions. You call the API that is defined by the transaction manager, not the XA functions listed below.

The transaction managers typically control the transactions through the TX interface. It includes the following functions:

<code>tx_open</code>	Logs into the resource manager(s)
<code>tx_close</code>	Logs out of the resource manager(s)
<code>tx_begin</code>	Starts a new transaction
<code>tx_commit</code>	Commits a transaction
<code>tx_rollback</code>	Rolls back the transaction

Most TPM applications are written using a client-server architecture where an application client requests services and an application server provides services. The examples that follow use such a client-server model. A service is a logical unit of work, which in the case of the Oracle Server as the resource manager, comprises a set of SQL statements that perform a related unit of work.

For example, when a service named "credit" receives an account number and the amount to be credited, it executes SQL statements to update information in certain tables in the database. In addition, a service might request other services. For example, a "transfer fund" service might request services from a "credit" and "debit" service.

Usually application clients request services from the application servers to perform tasks within a transaction. However, for some TPM systems, the application client itself can offer its own local services.

You can encode transaction control statements within either the client or the server; as shown in the examples.

To have more than one process participating in the same transaction, the TPM provides a communication API that allows transaction information to flow between the participating processes. Examples of communications APIs include RPC, pseudo-RPC functions, and send/receive functions.

Because the leading vendors support different communication functions, the examples that follow use the communication pseudo-function `tpm_service` to generalize the communications API.

X/Open has included several alternative methods for providing communication functions in their preliminary specification. At least one of these alternatives is supported by each of the leading TPM vendors.

## Examples of Precompiler Applications

The following examples illustrate precompiler applications. Assume that the application servers have already logged onto the TPM system, in a TPM-specific manner.

The first example shows a transaction started by an application server, and the second example shows a transaction started by an application client.

### Example 1: Transaction started by an application server

#### Client:

```
tpm_service("ServiceName");           /*Request Service*/
```

#### Server:

```
ServiceName()  
{  
<get service specific data>  
tx_begin();                           /* Begin transaction boundary*/  
EXEC SQL UPDATE ....;  
  
/*This application server temporarily becomes*/  
/*a client and requests another service.*/  
  
tpm_service("AnotherService");  
tx_commit();                           /*Commit the transaction*/  
<return service status back to the client>  
}
```

### Example 2: Transaction started by an application client.

#### Client:

```
tx_begin();                             /* Begin transaction boundary */  
tpm_service("Service1");  
tpm_service("Service2");  
tx_commit();                             /* Commit the transaction */
```

#### Server:

```
Service1()  
{  
<get service specific data>  
EXEC SQL UPDATE ....;  
<return service status back to the client>
```

```

}
Service2()
{
<get service specific data>
EXEC SQL UPDATE ....;
...
<return service status back to client>
}

```

## Migrating Precompiler or OCI Applications to TPM Applications

To migrate existing precompiler or OCI applications to a TPM application using the Oracle XA library, you must do the following:

1. Reorganize the application into a framework of "services".

This means that application clients request services from application servers.

Some TPMs require the application to use the `tx_open` and `tx_close` functions, whereas other TPMs do the logon and logoff implicitly.

If you do not specify the `sqlnet` parameter in your open string, then the application uses the default Net8 driver. Thus, you must be sure that the application server is brought up with the `ORACLE_HOME` and `ORACLE_SID` environment variables properly defined. This is accomplished in a TPM-specific fashion. See your TPM vendor documentation for instructions on how to accomplish this.

2. Ensure that the application replaces the regular connect and disconnect statements.

For example, replace the connect statements `EXEC SQL CONNECT` (for precompilers) or `OCISessionBegin()` (for OCIs) by `tx_open()`. Replace the disconnect statements `EXEC SQL COMMIT/ROLLBACK RELEASE WORK` (for precompilers), or `OCISessionEnd()` (for OCIs) by `tx_close()`. The V7 equivalent for `OCISessionBegin()` was `olon()` and for `OCISessionEnd()`, `ologof()`.

3. Ensure that the application replaces the regular commit/rollback statements and begins the transaction explicitly.

For example, replace the commit/rollback statements `EXEC SQL COMMIT/ROLLBACK WORK` (for precompilers), or `ocom()/orol()` (for OCIs) by `tx_commit()/tx_rollback()` and start the transaction by calling `tx_begin()`.

4. Ensure that the application resets the fetch state prior to ending a transaction. In general, `release_cursor=no` should be used. Use `release_cursor=yes` only when you are certain that a statement will be executed only once.

**Table 16–1** lists the TPM functions that replace regular Oracle commands when migrating precompiler or OCI applications to TPM applications.

**Table 16–1 TPM Replacement Commands**

Regular Oracle Commands	TPM Functions
CONNECT <i>user/password</i>	tx_open (possibly implicit)
implicit start of transaction	tx_begin
SQL	Service that executes the SQL
COMMIT	tx_commit
ROLLBACK	tx_rollback
disconnect	tx_close (possibly implicit)
SET TRANSACTION READ ONLY	Illegal

## XA Library Thread Safety

If you use a transaction monitor that supports threads, then the Oracle XA library allows you to write applications that are thread safe. Certain issues must be kept in mind, however.

A *thread of control* (or thread) refers to the set of connections to resource managers. In an unthreaded system, each process could be considered a thread of control, because each process has its own set of connections to resource managers and each process maintains its own independent resource manager table.

In a threaded system, each thread has an autonomous set of connections to resource managers and each thread maintains a *private* resource manager table. This private resource manager table must be allocated for each new thread and de-allocated when the thread terminates, even if the termination is abnormal.

---

---

**Note:** In an Oracle system, once a thread has been started and establishes a connection, only that thread can use that connection. No other thread can make a call on that connection.

---

---

### Specifying Threading in the Open String

The `xa_open` string parameter, `xa_info`, provides the clause, `Threads=`, which must be specified as `true` to enable the use of threads by the transaction monitor. The default is `false`. Note that, in most cases, threads are created by the transaction monitor, and the application does not know when a new thread is created. Therefore, it is advisable to allocate a service context (`lda` in Version 7) on the stack within each service that is written for a transaction monitor application. Before doing any Oracle-related calls in that service, the `xaoSvcCtx` (`sqlld2` for Version 7 OCI) function must be called and the service context initialized. This LDA can then be used for all OCI calls within that service.

### Restrictions on Threading in XA

The following restrictions apply when using threads:

- Any Pro\* or OCI code that executes as part of the application server process on the transaction monitor cannot be threaded unless the transaction monitor is explicitly told when each new application thread is started. This is typically accomplished by using a special C compiler provided by the transaction monitor vendor.
- The Pro\* statements, `EXEC SQL ALLOCATE` and `EXEC SQL USE` are not supported. Therefore, when threading is enabled, embedded SQL statements cannot be used across non-XA connections.
- If one thread in a process connects to Oracle through XA, all other threads in the process that connect to Oracle must also connect through XA. You cannot connect through `EXEC SQL` in one thread and through XA in another thread.

## Troubleshooting XA Applications

This section discusses how to find information in case of problems or system failure. It also discusses trace files and recovery of pending transactions.

### XA Trace Files

The Oracle XA library logs any error and tracing information to its trace file. This information is useful in supplementing the XA error codes. For example, it can indicate whether an `xa_open` failure is caused by an incorrect open string, failure to find the Oracle Server instance, or a logon authorization failure.

The name of the trace file is:

```
xa_db_namdate.trc
```

where *db\_name* is the database name you specified in the open string field `DB=db_name`, and *date* is the date when the information is logged to the trace file.

If you do not specify `DB=db_name` in the open string, then it automatically defaults to the name `NULL`.

### The `xa_open` string `DbgFl`

Normally, the XA trace file is opened only if an error is detected. The `xa_open` string `DbgFl` provides a tracing facility to record additional detail about the XA library. By default, its value is zero. It can be set to any combination of the following values. Note that they are independent, so to get printout from two or more flags, each must be set.

- `0x1` Trace the entry and exit to each procedure in the XA interface. This can be useful in seeing exactly what XA calls the TP Monitor is making and what transaction identifier it is generating.
- `0x2` Trace the entry to and exit from other non-public XA library routines. This is generally of use only to Oracle developers.
- `0x4` Trace various other "interesting" calls made by the XA library, such as specific calls to the Oracle Call Interface. This is generally of use only to Oracle developers.

### Trace File Locations

The trace file can be placed in one of the following locations:

- The trace file can be created in the `LogDir` directory as specified in the open string.
- If you do not specify `LogDir` in the open string, then the Oracle XA application attempts to create the trace file in the `$ORACLE_HOME/rdbms/log` directory, if it can determine where `$ORACLE_HOME` is located.
- If the Oracle XA application cannot determine where `$ORACLE_HOME` is located, then the trace file is created in the current working directory.

## Trace File Examples

Examples of two types of trace files are discussed below:

The example, `xa_NULL04021992.trc`, shows a trace file that was created on April 2, 1992. Its `DB` field was not specified in the open string when the resource manager was opened.

The example, `xa_Finance12151991.trc`, shows a trace file was created on December 15, 1991. Its `DB` field was specified as "Finance" in the open string when the resource manager was opened.

---

---

**Note:** Multiple Oracle XA library resource managers with the same `DB` field and `LogDir` field in their open strings log all trace information that occurs on the same day to the same trace file.

---

---

Each entry in the trace file contains information that looks like this:

```
1032.12345.2: ORA-01017: invalid username/password; logon denied
1032.12345.2: xaolgn: XAER_INVAL; logon denied
```

Where "1032" is the time when the information is logged, "12345" is the process ID (PID), "2" is the resource manager ID, `xaolgn` is the module name, `XAER_INVAL` was the error returned as specified in the XA standard, and `ORA-1017` is the Oracle Server information that was returned.

## In-doubt or Pending Transactions

In-doubt or pending transactions are transactions that have been prepared, but not yet committed to the database.

Generally, the transaction manager provided by the TPM system should resolve any failure and recovery of in-doubt or pending transactions. However, the DBA may

have to override an in-doubt transaction in certain circumstances, such as when the in-doubt transaction is:

- Locking data that is required by other transactions
- Not resolved in a reasonable amount of time

For more information about overriding in-doubt transactions in the circumstances described above, or about how to decide whether the in-doubt transaction should be committed or rolled back, see the TPM documentation.

## Oracle Server SYS Account Tables

There are four tables under the Oracle Server *SYS* account that contain transactions generated by regular Oracle Server applications and Oracle XA applications. They are `DBA_PENDING_TRANSACTIONS`, `V$GLOBAL_TRANSACTIONS`, `DBA_2PC_PENDING` and `DBA_2PC_NEIGHBORS`

For transactions generated by Oracle XA applications, the following column information applies specifically to the `DBA_2PC_NEIGHBORS` table.

- The `DBID` column is always `xa_orcl`
- The `DBUSER_OWNER` column is always `db_name.xa.oracle.com`

Remember that the *db\_name* is always specified as `DB=db_name` in the open string. If you do not specify this field in the open string, then the value of this column is `NULLxa.oracle.com` for transactions generated by Oracle XA applications.

For example, you could use the SQL statement below to obtain more information about in-doubt transactions generated by Oracle XA applications.

```
SELECT * FROM Dba_2pc_pending p, Dba_2pc_neighbors n
WHERE p.Local_tran_id = n.Local_tran_id
AND
n.Dbid = 'xa_orcl';
```

Alternatively, if you know the format ID used by the transaction processing monitor, then you can use `DBA_PENDING_TRANSACTIONS` or `V$GLOBAL_TRANSACTIONS`. While `DBA_PENDING_TRANSACTIONS` gives a list of both active and failed prepared transactions, `V$GLOBAL_TRANSACTIONS` gives a list of all active global transactions.

---

## General XA Issues and Restrictions

### Database Links

Oracle XA applications can access other Oracle Server databases through database links, with the following restrictions:

- Use the Multi-Threaded Server configuration.

This means that the transaction processing monitors (TPMs) use shared servers to open the connection to Oracle. The O/S network connection required for the database link is opened by the dispatcher, instead of the Oracle server process. Thus, when a particular service or RPC completes, the transaction can be detached from the server so that it can be used by other services or RPCs.

- Access to the other database must use SQL\*Net Version 2 or Net8.
- The other database being accessed should be another Oracle Server database.

Assuming that these restrictions are satisfied, Oracle Server allows such links and propagates the transaction protocol (prepare, rollback, and commit) to the other Oracle Server databases.

---

---

**Caution:** If these restrictions are not satisfied, then when you use database links within an XA transaction, it creates an O/S network connection in the Oracle Server that is connected to the TPM server process. Because this O/S network connection cannot be moved from one process to another, you cannot detach from this server. When you access a database through a database link, you receive an ORA#24777 error.

---

---

If using the Multi-Threaded Server configuration is not possible, then access the remote database through the Pro\*C/C++ application using EXEC SQL AT syntax.

The parameter `open_links_per_instance` specifies the number of migratable open database link connections. These `dblink` connections are used by XA transactions so that the connections are cached after a transaction is committed. Another transaction is free to use the `dblink` connection provided the user that created the connection is the same as the user who created the transaction. This parameter is different from the `open_links` parameter, which is the number of `dblink` connections from a session. The `open_links` parameter is not applicable to XA applications.

## Oracle Parallel Server Option

You can recover failed transactions from any instance of Oracle Parallel Server. You can also heuristically commit in-doubt transactions from any instance. An XA recover call gives a list of all prepared transactions for all instances.

## SQL-based Restrictions

### Rollbacks and Commits

Because the transaction manager is responsible for coordinating and monitoring the progress of the global transaction, the application should not contain any Oracle Server-specific statement that independently rolls back or commits a global transaction. However, you can use rollbacks and commits in a local transaction.

Do not use `EXEC SQL ROLLBACK WORK` for precompiler applications when you are in the middle of a global transaction. Similarly, an OCI application should not execute `OCITransRollback()`, or the Version 7 equivalent `orol()`. You can roll back a global transaction by calling `tx_rollback()`.

Similarly, a precompiler application should not have the `EXEC SQL COMMIT WORK` statement in the middle of a global transaction. An OCI application should not execute `OCITransCommit()` or the Version 7 equivalent `ocom()`. Instead, use `tx_commit()` or `tx_rollback()` to end a global transaction.

### DDL Statements

Because a DDL SQL statement, such as `CREATE TABLE`, implies an implicit commit, the Oracle XA application cannot execute any DDL SQL statements.

### Session State

Oracle does not guarantee that session state will be valid between services. For example, if a service updates a session variable (such as a global package variable), then another service that executes as part of the same global transaction may not see the change. Use savepoints only within a service. The application must not refer to a savepoint that was created in another service. Similarly, an application must not attempt to fetch from a cursor that was executed in another service.

### SET TRANSACTION

Do not use the `SET TRANSACTION READ ONLY | READ WRITE | USE ROLLBACK SEGMENT SQL` statement.

## Connecting or Disconnecting with EXEC SQL

Do not use the EXEC SQL command to connect or disconnect. That is, do not use EXEC SQL COMMIT WORK RELEASE or EXEC SQL ROLLBACK WORK RELEASE.

## Miscellaneous XA Issues

Note the following additional information about Oracle XA:

### Transaction Branches

Oracle Server transaction branches within the same global transaction can share locks in either a tightly or loosely coupled manner. However, if the branches are on different instances when running Oracle Parallel Server, then they will be loosely coupled.

In tightly coupled transaction branches, the locks are shared between the transaction branches. This means that updates performed in one transaction branch can be seen in other branches that belong to the same global transaction before the update is committed. The Oracle Server obtains the DX lock before executing any statement in a tightly coupled branch. Hence, the advantage of using loosely coupled transaction branches is that there is more concurrency (because a lock is not obtained before the statement is executed). The disadvantage is that all the transaction branches must go through the two phases of commit, that is, XA one phase optimization cannot be used. These trade-offs between tightly coupled branches and loosely coupled branches are illustrated in [Table 16-2](#).

**Table 16-2 Tightly and Loosely Coupled Transaction Branches**

Attribute	Tightly Coupled Branches	Loosely Coupled Branches
Two Phase Commit	Read-only Optimization [prepare for all branches, commit for last branch]	Two phases [prepare and commit for all branches]
Serialization	Database Call	None

### Association Migration

The Oracle Server does not support association migration (a means whereby a transaction manager may resume a suspended branch association in another branch).

### **Asynchronous Calls**

The optional XA feature asynchronous XA calls is not supported.

### **Initialization Parameters**

Set the `transactions_init.ora` parameter to the expected number of concurrent global transactions.

The parameter `open_links_per_instance` specifies the number of migratable open database link connections. These `dblink` connections are used by XA transactions so that the connections are cached after a transaction is committed.

**See Also:** ["Database Links"](#) on page 16-29

### **Maximum Connections per Thread**

The maximum number of `xa_opens` per thread is now 32. Previously, it was 8.

### **Installation**

No scripts need be executed to use XA. It is necessary, however, to run the `xaview.sql` script to run Release 7.3 applications with the Oracle8 Server. Grant the `SELECT` privilege on `SYS.DBA_PENDING_TRANSACTIONS` to all users that connect to Oracle through the XA interface.

### **Compatibility**

The XA library supplied with Release 7.3 can be used with a Release 8.0 Oracle Server. You must use the Release 7.2 XA library with a Release 7.2 Oracle Server. You can use the 8.0 library with a Release 7.3 Oracle Server. There is only one case of backward compatibility: an XA application that uses Release 8.0 OCI works with a Release 7.3 Oracle Server, but only if you use `sqlld2` and obtain an `lda_def` before executing SQL statements. Client applications must remember to convert the Version 7 LDA to a service handle using `OCILdaToSvcCtx()` after completing the OCI calls.

## **Changes to Oracle XA Support**

### **XA Changes from Release 8.0 to Release 8.1**

There are no changes for Release 8.1.

## XA Changes from Release 7.3 to Release 8.0

The following changes have been made:

- [Session Caching Is No Longer Needed](#)
- [Dynamic Registration Is Supported](#)
- [Loosely Coupled Transaction Branches Are Supported](#)
- [SQLLIB Is Not Needed for OCI Applications](#)
- [No Installation Script Is Needed to Run XA](#)
- [The XA Library Can Be Used with the Oracle Parallel Server Option on All Platforms](#)
- [Transaction Recovery for Oracle Parallel Server Has Been Improved](#)
- [Both Global and Local Transactions Are Possible](#)
- [The xa\\_open String Has Been Modified](#)

### Session Caching Is No Longer Needed

Session caching is unnecessary with the new OCI. Therefore, the old `xa_open` string parameter, `SesCacheSz`, has been eliminated. Consequently, you can also reduce the `sessions init.ora` parameter. Instead, set the `transactions init.ora` parameter to the expected number of concurrent global transactions. Because sessions are not migrated when global transactions are resumed, applications must not refer to any session state beyond the scope of a service.

For information on how to organize your application into services, refer to the documentation provided with the transaction processing monitor. In particular, savepoints and cursor fetch state are cancelled when a transaction is suspended. This means that a savepoint taken by the application in a service is invalid in another service, even though the two services may belong to the same global transaction.

### Dynamic Registration Is Supported

Dynamic registration can be used if both the XA application and the Oracle Server are Version 8.

**See Also:** ["Extensions to the XA Interface"](#) on page 16-8

### **Loosely Coupled Transaction Branches Are Supported**

The Oracle8 Server supports both loosely and tightly coupled transaction branches in a single Oracle instance. The Oracle7 Server supported only tightly coupled transaction branches in a single instance, and loosely coupled transaction branches in different instances.

### **SQLLIB Is Not Needed for OCI Applications**

OCI applications used to require the use of `SQLLIB`. This means that OCI programmers had to buy `SQLLIB`, even if they had no desire to develop Pro\* applications. This is no longer the case.

### **No Installation Script Is Needed to Run XA**

The SQL script `XAVIEW.SQL` is not needed to run XA applications in Oracle Version 8. It is, however, still necessary for Version 7.3 applications.

**See Also:** ["Responsibilities of the DBA or System Administrator"](#) on page 16-9

### **The XA Library Can Be Used with the Oracle Parallel Server Option on All Platforms**

It was not possible with Version 7 to use the Oracle XA library together with the Oracle Parallel Server option on certain platforms. (Only if the platform's implementation of the distributed lock manager supported transaction-based rather than process-based locking would the two work together.) This limitation is no longer the case; if you can run the Oracle Parallel Server option, then you can run the Oracle XA library.

### **Transaction Recovery for Oracle Parallel Server Has Been Improved**

All transactions can be recovered from any instance of Oracle Parallel Server. Use the `xa_recover` call to provide a snapshot of the pending transactions.

### **Both Global and Local Transactions Are Possible**

It is now possible to have both global and local transactions within the same XA connection. Local transactions are transactions that are completely coordinated by the Oracle Server. For example, the update below belongs to a local transaction.

```
CONNECT scott/tiger;
```

```
UPDATE Emp_tab SET Sal = Sal + 1; /* begin local transaction*/
COMMIT;                          /* commit local transaction*/
```

Global transactions, on the other hand, are coordinated by an external transaction manager such as a transaction processing monitor. In these transactions, the Oracle Server acts as a subordinate and processes the XA commands issued by the transaction manager. The update shown below belongs to a global transaction.

```
xa_open(oracle_xa+acc=p/SCOTT/TIGER+sestm=10", 1, TMNOFLAGS);
/* Transaction manager opens */
/* connection to the Oracle server*/
tpbegin(); /* begin global transaction, the transaction*/
/* manager issues XA commands to the oracle*/
/* server to start a global transaction */
UPDATE Emp_tab SET Sal = Sal + 1;
/* Update is performed in the */
/* global transaction*/
tpcommit(); /* commit global transaction, */
/* the transaction manager issues XA commands*/
/* to the Oracle server to commit */
/* the global transaction */
```

The Oracle7 Server forbids a local transaction from being started in an XA connection. The update shown below would return an ORA-2041 error code.

```
xa_open("oracle_xa+acc=p/SCOTT/TIGER+sestm=10" , 1, TMNOFLAGS);
/* Transaction manager opens */
/*connection to the Oracle server */
UPDATE Emp_tab SET Sal = Sal + 1; /* Oracle 7 returns an error */
```

The Oracle8 Server, on the other hand, allows local transactions to be started in an XA connection. The only restriction is that the local transaction must be ended (committed or rolled back) before starting a global transaction in the connection.

### The xa\_open String Has Been Modified

Two new parameters have been added. They are:

- Loose\_Coupling

This parameter has a Boolean value and should be set to false when connected to an Oracle7 Server. If set to true, then global transaction branches are loosely coupled; in other words, locks are not shared between branches.

- SesWt

This parameter's value indicates the time-out limit when waiting for a transaction branch that is being used by another session. If Oracle cannot switch to the transaction branch within `SesWt` seconds, then `XA_RETRY` is returned.

Two parameters have been made obsolete and should only be used when connected to an Oracle Server Release 7.3.

- `GPWD`

The group password is not used by Oracle8. A session that is logged in with the same user name as the session that created a transaction branch is allowed to switch to the transaction branch.

- `SesCacheSz`

This parameter is not used by Oracle8 because session caching has been eliminated.

## Symbols

---

- %ROWTYPE attribute, 9-7
  - used in stored functions, 9-8
- %TYPE attribute, 9-7

## A

---

- access
  - database, 11-28, 11-29
  - objects
    - sequences, 2-30
    - schema objects, 11-30, 11-31
    - remote integrity constraints, 4-14
    - triggers, 12-3, 12-48
- Active Data Object
  - translating to PSP, 15-3
- Active Server Pages
  - translating to PSP, 15-3
- ADD\_POLICY procedure, 11-38
- advantages
  - OCI, 1-7
- AFTER triggers
  - auditing and, 12-35, 12-38
  - correlation names and, 12-16
  - specifying, 12-7
- ALL\_ERRORS view
  - debugging stored procedures, 9-43
- ALL\_SOURCE view, 9-43
- allocation
  - extents, 5-17
- ALTER CLUSTER command, 2-6
  - ALLOCATE EXTENT option, 5-17
- ALTER INDEX command, 2-6
- ALTER privilege, 11-22
- ALTER SEQUENCE command, 2-30
- ALTER SESSION command
  - SERIALIZABLE, 7-25
  - SET SCHEMA, 11-19
- ALTER SESSION SET SCHEMA statement, 11-44
- ALTER TABLE command, 2-6, 2-9
  - defining integrity constraints, 4-18
  - DISABLE ALL TRIGGERS option, 12-32
  - DISABLE integrity constraint option, 4-21
  - DROP integrity constraint option, 4-24
  - ENABLE ALL TRIGGERS option, 12-31
  - ENABLE integrity constraint option, 4-21
  - INTRANS parameter, 7-25
- ALTER TRIGGER command
  - DISABLE option, 12-31
  - ENABLE option, 12-31
- altering
  - storage parameters, 2-9
  - tables, 2-9
- American National Standards Institute (ANSI)
  - ANSI-compatible locking, 7-17
- anonymous PL/SQL blocks, 11-10
  - about, 9-2
  - compared to triggers, 9-28
- ANSI SQL92
  - FIPS flagger, 7-2
- application context
  - as secure data cache, 11-47
  - bind variables, 11-48
  - creating, 11-52
  - design principles, 11-45
  - examples, 11-53
  - fine-grained access control, 11-14, 11-18, 11-47

- how to use, 11-49
- introduction, 11-4, 11-40
- parallel query, 11-51
- performance, 11-56
- returning predicate, 11-48
- security features, 11-41
- setting, 11-52
- USERENV namespace, 11-42
- using in policy, 11-52
- versioning, 11-51
- application security
  - considerations for use, 11-6
  - introduction, 11-4, 11-5
  - limitations, 11-15
  - specifying attributes, 11-41
  - through validation, 11-41
- applications
  - administrator, 11-8
  - calling stored procedures and packages, 9-51
  - database users, 11-6
  - One Big Application User model, 11-6, 11-7
  - roles, 11-5, 11-9
  - security, 11-7, 11-12
  - unhandled exceptions in, 9-46
- attributes, USERENV, 11-43
- auditing
  - compromised by One Big Application User, 11-6
  - n-tier systems, 11-66, 11-71
  - triggers and, 12-34
- authentication
  - compromised by One Big Application User, 11-6
  - middle tier, 11-4, 11-64, 11-65, 11-66, 11-67
  - n-tier systems, 11-69
- AUTHENTICATION\_DATA attribute, 11-44
- AUTHENTICATION\_TYPE attribute, 11-44
- autonomous routine, 7-32
- autonomous scope
  - versus autonomous transaction, 7-32
- autonomous transactions, 7-32 to 7-40
- AUTONOMOUS\_TRANSACTION, 7-32

## B

---

- BEFORE triggers
  - complex security authorizations, 12-48
  - correlation names and, 12-16
  - derived column values, 12-49
  - specifying, 12-7
- binary data
  - RAW and LONG RAW, 3-20
- bind variables, 11-48
- blank padding data
  - performance considerations, 3-6
- body
  - triggers, 12-14 to 12-20
- Boolean expressions, 3-27
- bulk binds, 9-24
  - DML statements, 9-25
  - FOR loops, 9-27
  - SELECT statements, 9-26
  - usage, 9-25
- BY REF phrase, 10-28

## C

---

- CACHE option
  - CREATE SEQUENCE command, 2-34
- caches
  - sequence cache, 2-33
  - sequence numbers, 2-29
- call specifications, 10-3 to 10-50
- callbacks, 10-44 to 10-46
- cancelling a cursor, 7-10
- CASCADE CONSTRAINTS option, 11-32
- CASCADE option
  - integrity constraints, 5-18
- CATPROC.SQL script, 12-4, 13-2
- CC date format, 3-12
- century, 3-10
  - date format masks, 3-9
- CGI variables, 15-2
- CHAR datatype, 3-5
  - column length, 3-6
  - increasing column length, 2-9
- CHARSETFORM property, 10-24
- CHARSETID property, 10-24

- CHARTOROWID function, 3-25
- CHECK constraint
  - triggers and, 12-40, 12-47
- check constraints
  - how to use, 4-15 to 4-20
- client events, 13-7
- CLIENT\_INFO attribute, USERENV, 11-43
- clusters, 5-14 to 5-19
  - allocating extents, 5-17
  - dropped tables and, 2-10
  - performance considerations, 5-15
- columns
  - accessing in triggers, 12-15
  - default values, 4-4
  - generating derived values with triggers, 12-49
  - increasing length, 2-9
  - listing in an UPDATE trigger, 12-6, 12-18
  - multiple foreign key constraints, 4-11
  - number of CHECK constraints limit, 4-16
- columns, privileges, 11-30, 11-31
- commands, SQL
  - when constraint checking occurs, 4-17
- COMMIT command, 7-5
- comparison operators
  - blank padding data, 3-6
  - comparing dates, 3-9
- compile-time errors, 9-41
- composite keys
  - restricting nulls in, 4-16
- concurrency, 7-22
- conditional predicates
  - trigger bodies, 12-14, 12-17
- consistency
  - read-only transactions, 7-8
- constraining tables, 12-22
- constraints
  - composite UNIQUE keys, 4-7
  - restriction on stored functions, 9-57
- conversion functions, 3-25
  - TO\_CHAR function, 3-12
  - TO\_CHAR function, year and century
    - considerations, 3-11
  - TO\_DATE function, 3-11
- converting data, 3-25
  - ANSI datatypes, 3-24
  - assignments, 3-25
  - expression evaluation, 3-27
  - SQL/DS and DB2 datatypes, 3-24
- cookies, 15-2
- correlation names, 12-13 to 12-17
  - NEW, 12-16
  - OLD, 12-16
  - REFERENCING option and, 12-17
  - when preceded by a colon, 12-16
- CREATE CLUSTER command, 2-6, 5-15
  - hash clusters, 5-19
  - HASH IS option, 5-20
  - HASHKEYS option, 5-20
- CREATE CONTEXT statement, 11-52
- CREATE INDEX command, 2-6, 5-5
  - ON CLUSTER option, 5-16
- CREATE PACKAGE BODY command, 9-15
- CREATE PACKAGE command, 9-15
- CREATE ROLE statement, 11-23
- CREATE SCHEMA command, 2-37
  - privileges required, 2-37
- CREATE SCHEMA statement, 11-19
- CREATE SEQUENCE command
  - CACHE option, 2-29, 2-34
  - examples, 2-34
  - NOCACHE option, 2-34
- CREATE SESSION statement, 11-19
- CREATE TABLE command, 2-3, 2-4, 2-6
  - CLUSTER option, 5-16
  - defining integrity constraints, 4-17
  - INITRANS parameter in, 7-25
- CREATE TRIGGER command, 12-3
  - REFERENCING option, 12-17
- CREATE VIEW command, 2-15
  - OR REPLACE option, 2-18
  - WITH CHECK OPTION, 2-15, 2-19
- creating
  - clusters, 5-15
  - hash clusters, 5-19
  - indexes, 5-5
  - integrity constraints, 4-2
  - multiple objects, 2-37
  - packages, 9-15
  - sequences, 2-34
  - synonyms, 2-35

- tables, 2-3, 2-4
- triggers, 12-3, 12-20
- views, 2-15
- cryptographic functions, 11-72
- CURRENT\_SCHEMA attribute, USERENV, 11-44
- CURRENT\_USER attribute, USERENV, 11-43
- CURRVAL pseudo-column, 2-31
  - restrictions, 2-32
- cursor variables, 9-38
  - declaring and opening, 9-38
- cursor, shared, 11-48
- cursors, 7-9
  - cancelling, 7-10
  - closing, 7-10
  - maximum number of, 7-9
  - pointers to, 9-38
  - private SQL areas and, 7-9

## D

---

- data blocks
  - factors affecting size of, 2-6
  - shown in ROWIDs, 3-22
- data control in OO4O, 1-19
- data conversion, 3-25
  - ANSI datatypes, 3-24
  - assignments, 3-25
  - expression evaluation, 3-27
  - SQL/DS and DB2 datatypes, 3-24
- data dictionary
  - compile-time errors, 9-42
  - dropped tables and, 2-10
  - integrity constraints in, 4-27
  - procedure source code, 9-43
  - schema object views, 2-41
- data encryption, 11-4, 11-72
- Data Encryption Standard (DES), 11-72
- data object number
  - extended ROWID, 3-21, 3-22
- database
  - application administrator, 11-8
  - applications and security, 11-5
  - global name in a distributed system, 2-38
  - security and schemas, 11-19
  - user and application user, 11-6

- database event notification, 13-1, 14-5
- datafiles
  - shown in ROWIDs, 3-22
- datatypes, 3-2
  - ANSI/ISO, 3-24
  - CHAR, 3-2, 3-5
  - choosing a character datatype, 3-5
  - column lengths for character types, 3-6
  - data conversion, 3-25
  - DATE, 3-8, 3-10
  - DB2, 3-24
  - LONG, 3-18
  - LONG RAW, 3-18, 3-20
  - NCHAR, 3-2, 3-5
  - NUMBER, 3-7
  - NVARCHAR2, 3-2, 3-5
  - RAW, 3-20
  - ROWID, 3-21
  - SQL/DS, 3-24
  - summary of datatypes, 3-2
  - VARCHAR, 3-5
  - VARCHAR2, 3-2, 3-5
- date arithmetic, 3-27
- DATE datatype, 3-8
  - centuries, 3-10
  - data conversion, 3-25
- DB\_DOMAIN attribute, USERENV, 11-44
- DBA\_ERRORS view
  - debugging stored procedures, 9-43
- DBA\_ROLE\_PRIVS view, 11-9
- DBA\_SOURCE view, 9-43
- DBMS\_LOCK package, 7-20
- DBMS\_OBFUSCATION\_TOOLKIT package, 11-72
- DBMS\_RLS package, 11-38, 11-46
- DBMS\_SESSION package
  - SET\_CONTEXT procedure, 11-52
  - SET\_ROLE procedure, 11-10
- DBMS\_SQL package
  - advantages of, 8-17
  - bulk SQL, 8-17
  - client-side programs, 8-17
  - DESCRIBE, 8-17
  - differences with native dynamic SQL, 8-12
  - multiple row updates and deletes, 8-18
  - RETURNING clause, 8-18



- errors
  - application errors raised by Oracle
    - packages, 9-43
  - creating views with errors, 2-16
  - remote procedures, 9-46
  - user-defined, 9-43, 9-44
- event attribute functions, 13-2
- event publication, 12-54 to 12-56, 13-1
  - triggering, 12-54
- event triggers, 11-59
- exception handlers
  - in PL/SQL, 9-2
- exceptions
  - anonymous blocks, 9-3
  - during trigger execution, 12-18
  - effects on applications, 9-46
  - remote procedures, 9-46
  - unhandled, 9-45
- exclusive locks
  - LOCK TABLE command, 7-16
- EXECUTE privilege, 11-22
- explicit locking
  - manual locking, 7-11
- extended ROWID format, 3-21
- extents
  - allocating, 5-17
  - dropped tabled and, 2-10
- external procedure, 10-2
  - DEBUG\_EXTPROC package, 10-48
  - debugging, 10-47
  - maximum number of parameters, 10-50
  - restrictions, 10-50
  - specifying datatypes, 10-16
- EXTERNAL\_NAME attribute, USERENV, 11-44
- extproc process, 10-33

## F

---

- fine-grained access control
  - application context, 11-14, 11-18, 11-45, 11-47
  - features, 11-36
  - introduction, 11-4, 11-35
  - performance, 11-37
- FIPS flagger
  - interactive SQL statements and, 7-2

- FIXED\_DATE initialization parameter, 3-9
- FOR EACH ROW clause, 12-12
- foreign key constraints
  - defining, 4-25, 4-26
  - enabling, 4-20, 4-27
  - NOT NULL constraint and, 4-10
  - one-to-many relationship, 4-10
  - one-to-n relationships, 4-10
  - UNIQUE key constraint and, 4-10
- format masks
  - TO\_DATE function, 3-8
- functions
  - See Also PL/SQL

## G

---

- GRANT command
  - ADMIN OPTION, 11-28
  - effective time, 11-34
  - last DDL time, 11-33
  - object privileges, 11-21, 11-30
  - system privileges, 11-28
- GRANT OPTION, 11-30
- grantee event attribute, 13-3
- granting
  - roles, 11-28
  - system privileges, 11-28, 11-29

## H

---

- hash clusters
  - how to use, 5-19 to 5-20
- HEXTORAW function, 3-25
- hiding PL/SQL code, 9-28
- hostname, 15-1
- HTML
  - displaying within PSP files, 15-5
  - retrieving from PL/SQL, 15-2
- HTP and HTF packages, 15-2
- HTTP URLs, 15-2

## I

---

- image maps, 15-2
- IN OUT parameter mode, 9-6

- IN parameter mode, 9-6
- INDEX privilege, 11-22
- indexes
  - creating, 5-5
  - dropped tables and, 2-10
  - dropping, 5-5
  - function-based, 5-6
  - guidelines, 5-2
  - order of columns, 5-4
  - privileges, 5-5
  - specifying PCTFREE for, 2-6
  - SQL\*Loader and, 5-2
  - temporary segments and, 5-2
  - when to create, 5-2
- index-organized tables, 6-1 to 6-20
- INDICATOR property, 10-23
- initialization parameters
  - DML\_LOCKS, 7-11
  - OPEN\_CURSORS, 7-9
  - REMOTE\_DEPENDENCIES\_MODE, 9-35
  - ROW\_LOCKING, 7-11
  - SERIALIZABLE, 7-11
- INTRANS parameter, 7-25
- INSERT command
  - column values and triggers, 12-16
  - read consistency, 7-11
- INSERT privilege, 11-22
- instance\_num event attribute, 13-3
- INSTEAD OF triggers, 12-7
  - on nested table view columns, 12-16
- integrity constraints
  - application uses, 4-2
  - clusters and, 5-16
  - defining, 4-17
  - disabling, 4-19, 4-20, 4-21
  - dropping, 4-24
  - enabling, 4-19
  - enabling when violations exist, 4-20
  - examples, 4-2
  - exceptions to, 4-22
  - listing definitions of, 4-27
  - naming, 4-18
  - performance considerations, 4-3
  - privileges required for creating, 4-18
  - triggers vs., 12-2, 12-40

- violations, 4-20
  - when to disable, 4-19
  - when to use, 4-2
- interactive block execution, 9-50
- invalid views, 2-20
- invoker's rights stored procedures, 11-10
- invoker-rights
  - dynamic SQL, 8-8
- is\_alter\_column event attribute, 13-3
- is\_creating\_nested\_table event attribute, 13-4
- is\_drop\_column event attribute, 13-4
- is\_servererror event attribute, 13-4
- ISDBA attribute, USERENV, 11-43
- ISOLATION LEVEL
  - changing, 7-25
  - SERIALIZABLE, 7-25

## J

---

- Java
  - calling methods through call specifications, 10-3
  - generating wrapper classes with
    - JPublisher, 1-33
  - in the RDBMS, 1-29
  - loading into the database, 10-4
  - overview of JDBC, 1-26
  - overview of SQLJ, 1-30
  - vs PL/SQL, 1-39
- Java Server Pages
  - translating to PSP, 15-3
- Javascript
  - translating to PSP, 15-3
- JDBC
  - See Oracle JDBC, 1-26
- join views, 2-21
  - DELETE statements, 2-24
  - key-preserved tables in, 2-22
  - mergeable, 2-22
  - modifying, 2-23
  - UPDATE statements, 2-24
  - when modifiable, 2-21
- JPublisher, 1-35
- JScript
  - translating to PSP, 15-3

## K

---

- key-preserved tables
  - in join views, 2-22
  - in outer joins, 2-27
- keys
  - foreign keys, 4-25
  - unique
    - composite, 4-7

## L

---

- libraries, 1-38
- library units
  - remote dependencies, 9-28
- Lightweight Directory Access Protocol (LDAP), 11-56
- lightweight sessions, 11-69
- loadjava utility, 1-36
- loadpsp command, 15-11
- LOB datatype
  - support in OO4O, 1-17
  - use in triggers, 12-20
- LOB support in OO4O, 1-17
- LOCK TABLE command, 7-12
- locking
  - manual (explicit), 7-11
- locks
  - distributed, 7-11
  - LOCK TABLE command, 7-12, 7-13
  - privileges for manual acquirement, 7-16
  - user locks, 7-20
  - UTLLOCKT.SQL script, 7-21
- login triggers, 11-49, 11-52, 11-53
- login\_user event attribute, 13-4
- LONG datatype, 3-18
  - restrictions on, 3-18
  - use in triggers, 12-20
- LONG RAW datatype, 3-18, 3-20
  - restrictions on, 3-18
  - use in triggers, 12-20
- LOWER function, 5-8

## M

---

mail

- sending from PL/SQL, 15-1
- manual locking, 7-11
  - LOCK TABLE command, 7-12
- MAX\_ENABLED\_ROLES parameter, 11-24, 11-27
- MAXTRANS option, 2-6
- memory
  - scalability, 9-70
- middle tier systems, 11-42, 11-64, 11-65, 11-66
- migration
  - ROWID format, 3-23
- modes
  - of parameters, 9-6
- modifiable join view
  - definition of, 2-21
- mutating tables, 12-22

## N

---

- name resolution, 2-38
- native dynamic SQL
  - advantages of, 8-13
  - differences with DBMS\_SQL package, 8-12
  - fetching into records, 8-16
  - performance, 8-15
  - See Also* dynamic SQL
  - user-defined types, 8-16
- NCHAR datatype, 3-2, 3-5
- NEW correlation name, 12-16
- NEXTVAL pseudo-column, 2-31
  - restrictions, 2-32
- NLS\_DATE\_FORMAT parameter, 3-8
- NLSSORT order, and indexes, 5-8
- NOCACHE option
  - CREATE SEQUENCE statement, 2-34
- NOT NULL constraint
  - CHECK constraint and, 4-16
  - data integrity, 4-20
  - when to use, 4-3
- NOWAIT option, 7-12
- n-tier authentication, 11-64, 11-65, 11-66
- NUMBER datatype, 3-7
- NVARCHAR2 datatype, 3-2, 3-5

## O

---

- OAS, 15-2
- object columns, indexes on, 5-7
- object support in OO4O, 1-17
- objects
  - GRANT OPTION, 11-30
  - granting privileges, 11-22, 11-30
  - privileges, 11-21
  - revoking privileges, 11-31
- objects, schema
  - listing information, 2-41
  - name resolution, 2-38
  - renaming, 2-40
- OCI, 9-2
  - advantages, 1-7
  - applications, 9-4
  - cancelling cursors, 7-10
  - closing cursors, 7-10
  - enabling roles, 11-17
  - overview, 1-7
  - parts of, 1-8
- OCI vs precompilers, 1-37
- OLD correlation name, 12-16
- one-to-many relationship
  - with foreign keys, 4-10
- one-to-one relationship
  - with foreign keys, 4-10
- OO4O
  - See Oracle Objects for OLE, 1-12
- open string for XA, 16-10
- OPEN\_CURSORS parameter, 7-9
- operating system
  - roles, 11-27
- OR REPLACE clause
  - for creating packages, 9-16
- ora\_dictionary\_obj\_owner event attribute, 13-3
- ora\_dictionary\_obj\_owner\_list event attribute, 13-3
- ora\_dictionary\_obj\_type event attribute, 13-3
- ora\_grantee event attribute, 13-3
- ora\_instance\_num event attribute, 13-3
- ora\_is\_alter\_column event attribute, 13-3
- ora\_is\_creating\_nested\_table event attribute, 13-4
- ora\_is\_drop\_column event attribute, 13-4
- ora\_is\_servererror event attribute, 13-4
- ora\_login\_user event attribute, 13-4
- ora\_privileges event attribute, 13-4
- ora\_revokee event attribute, 13-4
- ora\_server\_error event attribute, 13-4
- ora\_sysevent event attribute, 13-4
- ora\_with\_grant\_option event attribute, 13-5
- ORA-21301 error, fixing, 16-15
- OraAQ object, 1-16
- OraAQAgent object, 1-17
- OraAQMsg object, 1-17
- OraBFILE object, 1-18
- OraBLOB object, 1-18
- Oracle Advanced Security, 11-19
- Oracle Application Server (OAS), 15-2
- Oracle Call Interface
  - See OCI
- Oracle Data Control (ODC), 1-19
- Oracle errors, 9-3
- Oracle Internet Directory, 11-67
- Oracle JDBC
  - definition, 1-26
  - example, 1-27
  - OCI driver, 1-26
  - Oracle extensions, 1-27
  - server driver, 1-27
  - stored procedures, 1-29
  - thin driver, 1-26
- Oracle Objects for OLE
  - automation server, 1-12
  - C++ Class Library, 1-19
  - data control, 1-19
  - LOB and object support, 1-17
  - object model, 1-12
  - overview, 1-11
- Oracle SQLJ
  - advantages over JDBC, 1-32
  - definition, 1-30
  - design, 1-31
  - example, 1-33
  - in the server, 1-36
  - stored programs, 1-36
- Oracle supplied packages, 9-17
- OraCLOB object, 1-18
- OraDatabase object, 1-14
- OraDynaset object, 1-14

- OraField object, 1-15
- OraMeta Data object, 1-15
- OraParamArray object, 1-16
- OraParameter object, 1-15
- OraServer object, 1-13
- OraSession object, 1-13
- OraSQLStmt object, 1-16
- OS\_ROLES parameter, 11-27
- OS\_USER attribute, USERENV, 11-44
- OUT parameter mode, 9-6
- outer joins, 2-26
  - key-preserved tables in, 2-27
- overloading
  - of packaged functions, 9-70
  - stored procedure names, 9-13
  - using RESTRICT\_REFERENCES, 9-70
- OWA\* packages, 15-2

## P

---

- package body, 9-12
- package specification, 9-12
- packages, 1-38
  - creating, 9-15
  - DBMS\_OUTPUT
    - example of use, 9-3
  - DEBUG\_EXTPROC, 10-48
  - dropping, 9-12
  - in PL/SQL, 9-12
  - naming of, 9-16
  - privileges for execution, 9-52
  - privileges required to create, 9-16
  - privileges required to create procedures in, 9-10
  - serially reusable packages, 9-70
  - session state and, 9-17
  - synonyms, 9-56
  - where documented, 9-17
- parallel execution servers, 11-51
- parallel query, and SYS\_CONTEXT, 11-51
- parallel server
  - distributed locks, 7-11
  - sequence numbers and, 2-30
- PARALLEL\_ENABLE keyword, 9-63
- parameters
  - default values, 9-9

- with stored functions, 9-59
- modes, 9-6
- parse tree, 12-28
- Pascal Calling Standard, 10-8
- passwords
  - role, 11-18, 11-25
- pcode
  - when generated for triggers, 12-28
- PCTFREE storage parameter, 2-4 to 2-9
- PCTUSED storage parameter, 2-6 to 2-9
  - guidelines for setting, 2-6
- performance
  - clusters, 5-15
  - index column order, 5-4
  - native dynamic SQL, 8-15
- PL/SQL, 9-2
  - advantages, 1-3
  - anonymous blocks, 9-2, 11-10
  - calling remote stored procedures, 9-54
  - cursor variables, 9-38
  - dependencies among library units, 9-28
  - dynamically modifying SQL statements, 11-14
  - exception handlers, 9-2
  - features, 1-3
  - functions
    - arguments, 9-59
    - overloading, 9-70
    - parameter default values, 9-59
    - purity level, 9-69
    - RESTRICT\_REFERENCES pragma, 9-66
    - using, 9-56
  - hiding source code, 9-28
  - invoking with dynamic SQL, 8-7
  - objects, 1-5
  - packages, 9-12
    - dropped tables and, 2-10
    - replaced views and, 2-18
  - RAISE statement, 9-44
  - reparsing, 11-45, 11-46
  - sample code, 1-2
  - serially reusable packages, 9-70
  - server pages, 15-2 to 15-12
  - setting context, 11-49
  - tables, 9-9

- of records, 9-9
  - trigger bodies, 12-14, 12-15
  - user-defined errors, 9-44
  - vs Java, 1-39
  - web toolkit, 15-2
  - wrapper to hide code, 9-28
- pragma, 7-32, 7-40
  - RESTRICT\_REFERENCES pragma, 9-66
  - SERIALLY\_REUSABLE pragma, 9-70, 9-71
- precompilers, 9-51
  - applications, 9-4
  - calling stored procedures and packages, 9-51
- precompilers vs OCI, 1-37
- PRIMARY KEY constraint
  - choosing a primary key, 4-6
  - disabling, 4-20
  - enabling, 4-20
  - multiple columns in, 4-7
  - UNIQUE key constraint vs., 4-7
- private SQL areas
  - cursors and, 7-9
- privileges
  - altering sequences, 2-30
  - altering tables, 2-10
  - cluster creation, 5-16
  - creating integrity constraints, 4-18
  - creating tables, 2-8
  - creating triggers, 12-26
  - dropping a view, 2-20
  - dropping sequences, 2-35
  - dropping tables, 2-11
  - dropping triggers, 12-30
  - encapsulating in stored procedures, 11-17
  - granting, 11-28, 11-30
  - granting to PUBLIC, 11-34
  - index creation, 5-5
  - managing, 11-8, 11-21
  - manually acquiring locks, 7-16
  - middle tier, 11-70
  - n-tier systems, 11-66
  - object, 11-22
  - on selected columns, 11-31
  - recompiling triggers, 12-29
  - renaming objects, 2-40
  - replacing views, 2-18

- revoking, 11-29, 11-31
  - sequence creation, 2-30
  - SQL statements permitted, 11-22
  - stored procedure execution, 9-52
  - synonym creation, 2-35
  - triggers, 12-26
  - using a view, 2-20
  - using sequences, 2-34
  - view creation, 2-17
- privileges event attribute, 13-4
- Pro\*C/C++
  - overview of application development, 1-20 to 1-22
- Pro\*COBOL
  - overview of application development, 1-23 to 1-25
- procedures
  - called by triggers, 12-20
  - external, 10-2
- PRODUCT\_USER\_PROFILE table, 11-13, 11-14, 11-17
- program units in PL/SQL, 9-2
- property
  - CHARSETFORM, 10-24
  - CHARSETID, 10-24
  - INDICATOR, 10-23
- PROXY\_USER attribute, 11-42, 11-44
- pseudocolumns
  - modifying views, 12-8
- PSP
  - See PL/SQL server pages
- .psp files, 15-4
- PUBLIC user group, 11-34
- publish-subscribe, 14-2 to 14-6
- purity level, 9-61

## Q

---

- queries
  - capturing as views, 2-15
  - dynamic, 8-4
  - errors in distributed queries, 9-46
  - speeding up with temporary tables, 2-11

## R

---

- RAISE statement, 9-44
- RAISE\_APPLICATION\_ERROR procedure, 9-43
  - remote procedures, 9-46
- raising exceptions
  - triggers, 12-18
- RAW datatype, 3-20
- RAWTOHEX function, 3-25
- read-only transactions, 7-8
- reauthenticating clients, 11-67, 11-68, 11-71
- REF column
  - indexes on, 5-7
- REFERENCES privilege, 11-22, 11-32
- REFERENCING option, 12-17
- referential integrity
  - distributed databases and, 4-14
  - one-to-many relationship, 4-10
  - one-to-one relationship, 4-10
  - privileges required to create foreign keys, 4-25
  - self-referential constraints, 12-44
  - triggers and, 12-41 to 12-45
- REFRESH\_POLICY procedure, 11-38, 11-46
- remote dependencies, 9-28
  - signatures, 9-30
  - specifying timestamps or signatures, 9-35
- remote exception handling, 9-46, 12-18
- REMOTE\_DEPENDENCIES\_MODE
  - parameter, 9-35
- RENAME command, 2-39, 2-40
- REPARSE statement, 11-46
- reparsing, 11-37, 11-45, 11-46, 11-52
- repeatable reads, 7-8, 7-11
- resource manager, 16-2
- resource manager events, 13-6
- RESOURCE privilege, 11-19
- RESTRICT\_REFERENCES pragma
  - syntax for, 9-66
  - using to control side effects, 9-66
- restrictions
  - system triggers, 12-25
- reusable packages, 9-70
- REVOKE command, 11-29, 11-34
- revoker event attribute, 13-4
- revoking
  - privileges on selected columns, 11-31
  - roles and privileges, 11-29
- RM (resource manager), 16-2
- RNDS argument, 9-66
- RNPS argument, 9-66
- ROLE\_SYS\_PRIVS view, 11-9
- ROLE\_TAB\_PRIVS view, 11-9
- roles
  - ADMIN OPTION, 11-29
  - advantages, 11-8
  - application, 11-9, 11-12, 11-21
  - central management, 11-23
  - creating, 11-23
  - default, 11-24
  - dropping, 11-27
  - enabling, 11-9
  - enabling and disabling, 11-16
  - GRANT and REVOKE commands, 11-27
  - granting, 11-28
  - granting to PUBLIC, 11-34
  - management, 11-8
  - managing, 11-21
  - operating system, 11-27
  - passwords, 11-18, 11-25
  - protecting, 11-23
  - recommended practices, 11-16
  - restricting from tool users, 11-12
  - revoking, 11-29
  - SET ROLE statement, 11-27
  - system privileges, 11-28
  - usefulness compromised, 11-6
  - user, 11-9, 11-21
  - WITH GRANT OPTION, 11-31
- ROLLBACK command, 7-6
- rolling back transactions
  - to savepoints, 7-6
- routines
  - autonomous, 7-32
  - external, 10-2
  - service, 10-35
- row locking
  - manually locking, 7-18
- row triggers
  - defining, 12-12
  - REFERENCING option, 12-17

- timing, 12-7
- UPDATE statements and, 12-6, 12-18
- ROW\_LOCKING parameter, 7-11
- ROWID datatype, 3-21
  - extended ROWID format, 3-21
  - migration, 3-23
- ROWIDTOCHAR function, 3-25
- rows
  - chaining across blocks, 2-5
  - format, 2-2
  - header, 2-2
  - shown in ROWIDs, 3-22
  - size, 2-2
  - violating integrity constraints, 4-20
- ROWTYPE\_MISMATCH exception, 9-41
- RR date format, 3-11
- RS locks
  - LOCK TABLE command, 7-13
- run-time error handling, 9-43
- RX locks
  - LOCK TABLE command, 7-13

## S

---

- S locks
  - LOCK TABLE command, 7-13
- SAVEPOINT command, 7-6
- savepoints
  - maximum number of, 7-6
  - rolling back to, 7-6
- scalability
  - serially reusable packages, 9-70
- schema-independent users, 11-19, 11-20
- schemas
  - default, 11-44
  - unique, 11-19
- scope, autonomous, 7-32
- scripting, 15-2
- Secure Sockets Layer (SSL) protocol, 11-68
- security
  - application context, 11-40
  - enforcement in application, 11-7
  - enforcement in database, 11-7
  - features of Oracle8i, 11-4
  - fine-grained access control, 11-35

- policy for applications, 11-5, 11-12
- roles, advantages, 11-8
- table- or view-based, 11-35
- threats and countermeasures, 11-2
- security policies
  - administering, 11-38
  - applied within database, 11-15
  - centrally managed, 11-14
  - establishing, 11-1, 11-5
  - example, 11-39
  - implementing, 11-47
  - multiple policies per table, 11-37
  - on tables or views, 11-36
  - technical issues, 11-3
  - threats and countermeasures, 11-2
- SELECT command
  - read consistency, 7-11
  - SELECT ... FOR UPDATE, 7-18
- SELECT privilege, 11-22
- SEQUENCE\_CACHE\_ENTRIES parameter, 2-33
- sequences
  - accessing, 2-30
  - altering, 2-30
  - caching numbers, 2-29
  - caching sequence numbers, 2-33
  - creating, 2-29, 2-34
  - CURRVAL, 2-30, 2-32
  - dropping, 2-34
  - initialization parameters, 2-29
  - NEXTVAL, 2-31
  - parallel server, 2-30
  - privileges needed, 2-30 to 2-35
  - reducing serialization, 2-31
- SERIALIZABLE option
  - for ISOLATION LEVEL, 7-25
- SERIALIZABLE parameter, 7-11
- serializable transactions, 7-22
- serially reusable PL/SQL packages, 9-70
- SERIALLY\_REUSABLE pragma, 9-71
- server\_error event attribute, 13-4
- service routine, 10-35
  - examples, 10-35
- session primitives, 11-42
- SESSION\_USER attribute, USERENV, 11-43
- sessions

- package state and, 9-17
- SET ROLE statement
  - ALL EXCEPT option, 11-26
  - ALL option, 11-26
  - associating privileges with role, 11-9
  - at startup, 11-16
  - disabling, 11-17
  - enabling roles, 11-25
  - equivalent to SET\_ROLE, 11-10
  - operating system roles, 11-27
  - protecting role use, 11-23
  - role passwords, 11-18
- SET TRANSACTION command, 7-8
  - ISOLATION LEVEL clause, 7-25
  - SERIALIZABLE, 7-25
- SET\_CONTEXT procedure, 11-52
- SET\_ROLE procedure, 11-10
- SGA
  - See Also system global area
- share locks (S)
  - LOCK TABLE command, 7-13
- share row exclusive locks (SRX)
  - LOCK TABLE command, 7-15
- side effects, 9-6, 9-61
- signatures
  - PL/SQL library unit dependencies, 9-28
  - to manage remote dependencies, 9-30
- SORT\_AREA\_SIZE parameter
  - index creation and, 5-2
- sorting
  - with function-based indexes, 5-6
- SQL statements
  - dynamic, 11-50
  - execution, 7-2
  - in trigger bodies, 12-15, 12-20
  - not allowed in triggers, 12-20
  - privileges required for, 11-22
  - restricting ad hoc use, 11-12, 11-13
- SQL\*Loader
  - indexes and, 5-2
- SQL\*Module
  - applications, 9-4
- SQL\*Plus
  - anonymous blocks, 9-4
  - compile-time errors, 9-41
  - invoking stored procedures, 9-49
  - loading a procedure, 9-10
  - restricting ad hoc use, 11-12, 11-13
  - SET SERVEROUTPUT ON command, 9-3
  - SHOW ERRORS command, 9-41
- SQLStmt object, 1-16
- SRX locks
  - LOCK TABLE command, 7-15
- standards
  - ANSI, 7-17
- state
  - session, of package objects, 9-17
- statement triggers
  - conditional code for statements, 12-17
  - row evaluation order, 12-21
  - specifying SQL statement, 12-5
  - timing, 12-7
  - trigger evaluation order, 12-21
  - UPDATE statements and, 12-6, 12-18
  - valid SQL statements, 12-20
- storage parameters
  - PCTFREE, 2-9
  - PCTUSED, 2-9
- stored functions, 9-5
  - creating, 9-9
- stored procedures, 9-5
  - argument values, 9-52
  - creating, 9-9
  - distributed query creation, 9-46
  - encapsulating privileges, 11-17
  - exceptions, 9-43, 9-44
  - invoker's rights, 11-10
  - invoking, 9-49
  - names of, 9-5
  - overloading names of, 9-13
  - parameter
    - default values, 9-9
  - privileges, 9-52
  - remote, 9-53
  - remote objects and, 9-54
  - storing, 9-9
  - synonyms, 9-56
  - turning into a web page, 15-2
- synonyms
  - dropped tables and, 2-10

- how to use, 2-35 to 2-36
- stored procedures and packages, 9-56
- SYS schema, 11-52
- SYS\_CONTEXT function
  - access control, 11-58
  - dynamic SQL statements, 11-50
  - parallel query, 11-51
  - storing session variables, 11-51
  - syntax, 11-50
  - USERENV namespace, 11-42
- SYSDATE function, 3-9
- sysevent event attribute, 13-4
- system events, 13-1
  - attributes, 13-2
  - client events, 13-7
  - resource manager events, 13-6
  - tracking, 12-52, 13-1
- system global area
  - holds sequence number cache, 2-33
- system privileges, 11-28, 11-29
- system-specific Oracle documentation, 12-4
  - PL/SQL wrapper, 9-28

## T

---

table- or view-based security, 11-35

### tables

- altering, 2-9
- constraining, 12-22
- creating, 2-3, 2-4
- designing, 2-3
- dropping, 2-10
- guidelines, 2-2, 2-4
- in PL/SQL, 9-9
- increasing column length, 2-9
- index-organized, 6-1 to 6-20
- key-preserved, 2-22
- location, 2-4
- mutating, 12-22
- privileges for creation, 2-8
- privileges for dropping, 2-11
- privileges to alter, 2-10
- schema of clustered, 5-16
- specifying PCTFREE for, 2-5
- specifying PCTUSED for, 2-6

- specifying tablespace, 2-4
- truncating, 2-10
- TCP/IP, 15-1
- temporary segments
  - index creation and, 5-2
- TERMINAL attribute, USERENV, 11-43
- third generation language, 9-2
- three-tier systems, 11-64, 11-65, 11-66
- timestamps
  - PL/SQL library unit dependencies, 9-28
- TM (transaction manager), 16-2
- TO\_CHAR function, 3-25
  - CC date format, 3-12
  - RR date format, 3-6
- TO\_DATE function, 3-8, 3-25
  - RR date format, 3-11
- TO\_NUMBER function, 3-25
- transaction manager, 16-2
- transactions
  - autonomous, 7-32 to 7-40
  - manual locking, 7-11
  - read-only, 7-8
  - serializable, 7-22
  - SET TRANSACTION command, 7-8
- triggers
  - about, 9-28
  - accessing column values, 12-15
  - AFTER, 12-7, 12-16, 12-35, 12-38
  - auditing with, 12-34, 12-36
  - BEFORE, 12-7, 12-16, 12-48, 12-49
  - body, 12-14, 12-17, 12-18, 12-20
  - check constraints, 12-47, 12-48
  - client events, 13-7
  - column list in UPDATE, 12-6, 12-18
  - compiled, 12-28
  - conditional predicates, 12-14, 12-17
  - CREATE TRIGGER ON, 11-22
  - creating, 12-3, 12-20, 12-26
  - data access restrictions, 12-48
  - debugging, 12-30
  - designing, 12-2
  - disabling, 12-31
  - distributed query creation, 9-46
  - dropped tables and, 2-10
  - enabling, 12-31

- error conditions and exceptions, 12-18
- event, 11-59
- events, 12-5
- examples, 12-34 to 12-50
- FOR EACH ROW clause, 12-12
- generating derived column values, 12-49
- illegal SQL statements, 12-20
- INSTEAD OF triggers, 12-7
- integrity constraints vs., 12-2, 12-40
- listing information about, 12-32
- login, 11-49, 11-52, 11-53
- migration issues, 12-29
- modifying, 12-30
- multiple same type, 12-21
- mutating tables and, 12-22
- naming, 12-5
- package variables and, 12-21
- prerequisites before creation, 12-4
- privileges, 12-26
  - to drop, 12-30
- procedures and, 12-20
- recompiling, 12-29
- REFERENCING option, 12-17
- referential integrity and, 12-41 to 12-45
- remote dependencies and, 12-20
- remote exceptions, 12-18
- resource manager events, 13-6
- restrictions, 12-13, 12-20
- row, 12-12
- row evaluation order, 12-21
- scan order, 12-21
- stored, 12-28
- system triggers, 12-4
  - on DATABASE, 12-4
  - on SCHEMA, 12-4
- trigger evaluation order, 12-21
- use of LONG, LONG RAW, and LOB
  - datatypes, 12-20
- username reported in, 12-26
  - WHEN clause, 12-13
- TRUNC function, 3-9
- TRUNCATE TABLE command, 2-10
- TRUST keyword, 9-68
- tuning
  - using LONGs, 3-19

## U

---

- unhandled exceptions, 9-45
- UNIQUE key constraints
  - combining with NOT NULL constraint, 4-4
  - composite keys and nulls, 4-7
  - disabling, 4-20
  - enabling, 4-20
  - PRIMARY KEY constraint vs., 4-7
  - when to use, 4-7
- UPDATE command
  - column values and triggers, 12-16
  - data consistency, 7-11
  - triggers and, 12-6, 12-18
  - triggers for referential integrity, 12-43, 12-44
- UPDATE\_CHECK parameter, 11-38
- UPPER function, 5-8
- URLs, 15-2
- USER function, 4-5
- user locks
  - requesting, 7-20
- USER\_ERRORS view
  - debugging stored procedures, 9-42
- USER\_SOURCE view, 9-43
- user-defined errors, 9-43, 9-44
- USERENV function, 11-42
- USERENV namespace, 11-42, 11-43
- usernames
  - as reported in a trigger, 12-26
  - schemas, 11-19
- users
  - dropped roles, 11-27
  - enabling roles for, 11-9
  - enterprise, 11-19
  - PUBLIC group, 11-34
  - restricting application roles, 11-12
  - schema-independent, 11-19
- UTL\_HTTP package, 15-2
- UTL\_INADDR package, 15-1
- UTL\_SMTP package, 15-1
- UTL\_TCP package, 15-1
- UTLLOCKT.SQL script, 7-21

## V

---

validation function, 11-41  
VARCHAR datatype, 3-5  
VARCHAR2 datatype, 3-2, 3-5  
    column length, 3-6  
    when to use, 3-5  
VBScript  
    translating to PSP, 15-3  
versioning, in application context, 11-51  
views  
    containing expressions, 12-8  
    creating, 2-15  
    creating with errors, 2-16  
    dropped tables and, 2-10  
    dropping, 2-20  
    FOR UPDATE clause and, 2-15  
    inherently modifiable, 12-8  
    invalid, 2-20  
    join views, 2-21  
    modifiable, 12-8  
    ORDER BY clause and, 2-15  
    privileges, 2-17  
    pseudocolumns, 12-8  
    replacing, 2-17  
    restrictions, 2-19  
    using, 2-18  
    when to use, 2-15  
    WITH CHECK OPTION, 2-15  
virtual private database (VPD), 11-7, 11-14, 11-15,  
    11-17, 11-47

## W

---

web pages  
    dynamic, 15-2  
WebDB, 15-2  
WHEN clause, 12-13  
    cannot contain PL/SQL expressions, 12-13  
    correlation names, 12-16  
    examples, 12-3, 12-12, 12-32, 12-41  
    EXCEPTION examples, 12-18, 12-41, 12-47,  
        12-48  
WHERE clause, dynamic SQL, 11-14  
WITH CONTEXT clause, 10-28

with\_grant\_option event attribute, 13-5  
WNDS argument, 9-66  
WNPS argument, 9-66  
wrapper to hide PL/SQL code, 9-28

## X

---

X locks  
    LOCK TABLE command, 7-16  
XA library, 16-1 to 16-36  
xa\_open string, 16-10  
XML  
    as document type for PSP file, 15-5  
X/Open distributed transaction processing  
    architecture, 16-2

## Y

---

year 2000, 3-10

